A Scalable Ordering Primitive with Invariant Hardware Clocks

Paper #170

12 pages

Abstract

Timestamping is an essential building block for designing concurrency control mechanisms and concurrent data structures. Various algorithms either employ physical timestamping, assuming that they have access to synchronized clocks, or maintain a logical clock with the help of atomic instructions, a widely used approach among the algorithms. Unfortunately, these approaches have two problems. First, hardware developers do not guarantee that the available hardware clocks are exactly synchronized, which they find difficult to achieve in practice. Second, the atomic instructions are a deterrent to scalability resulting from cache-line contention. This paper addresses these problems by designing a scalable, ordering primitive, called ORDO, which relies on *invariant* hardware clocks. ORDO not only enables the correct use of these clocks but also frees other logical timestamp-based algorithms from the burden of the software logical clock. We use the ORDO primitive to redesign 1) a concurrent data structure library that we apply on the Linux kernel; 2) a synchronization mechanism for concurrent programming; 3) two database concurrency control mechanisms; and 4) a clock-based software transactional memory algorithm. Our evaluation shows 1) that there is a possibility that the clocks are not synchronized on two architectures (`Intel` and `ARM`), and 2) that ORDO improves the efficiency of concurrent algorithms by 1.2–39.7× on various architectures.

1. Introduction

Ordering is fundamental to the design of any concurrent algorithm whose purpose is to achieve varying levels of consistency. For various systems software, the requirement of consistency depends on algorithms that either require linearizability [28] for the composability of a data structure [9, 44], concurrency control mechanisms for software transactional memory (`STM`) [23] or for database transactions [36, 62, 67].

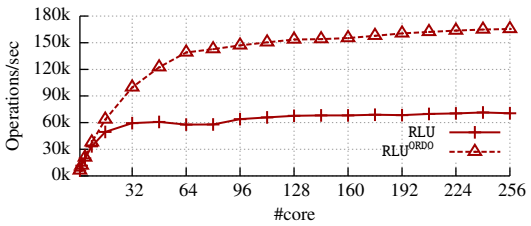

Figure 1: Throughput of read-log-update (RLU), a concurrency mechanism, on a hash table benchmark with 98% reads and 2% writes. While the original version (`RLU`) maintains a global logical clock, `RLU`$^{\text{ORDO}}$ is a redesign of `RLU` with our proposed ORDO primitive.

The notion of ordering is not only limited to consistency, but is also applicable to either maintaining the history of operations for logging [33, 39, 50, 64] or determining the quiescence period for memory reclamation [5, 44, 63]. Depending on the consistency requirement, ordering such as a logical clock [23, 44], physical timestamping [5, 9, 20, 24, 58], data-driven versioning [37, 67]can be achieved in several ways. The most common approach is to use logical clocks that are easier to maintain by software and is amenable to various ordering requirements.

A logical clock, a favorable ordering approach, is one of the prime scalability bottlenecks on large multicore and multi-socket machines [31, 53] because it is maintained via an atomic instruction that incurs cache-coherence traffic. Unfortunately, cache-line contention caused by atomic instructions becomes the scalability bottleneck, and it has become even more severe across the NUMA boundary and in hyper-threaded scenarios [13, 22, 59]. For example, a recently proposed synchronization mechanism for concurrent programming, called `RLU` [44], is saturated at eight cores for a mere 2% of writes on massively parallel cores of `Intel Xeon Phi` (Figure 1).[1] Thus, maintaining a software clock is a deterrent to the scalability of the application which holds true for other concurrency control mechanisms for concurrent programming [23, 26] and database transactions [36, 62]. To address this problem, various approaches ranging from batched-update [62] to local versioning [67] to completely share-nothing designs [10, 55] have been put into practice; however,

[1] `Intel Xeon Phi` has a slower processor, but has a higher memory bandwidth compared with other `Xeon`-based processors.

these approach have some side effects. They either increase the abort rates for STM [8] and databases [62], and even complicate the design of the systems software [10, 18, 67, 68]. In addition, in the past decade, physical timestamping has been gaining traction for designing scalable concurrent data structures [24, 27] and a synchronization mechanism [9] for large multicore machines, which assume that timestamp counters provided by the hardware are synchronized.

In this paper, our goal is to address the problem of a scalable timestamping primitive by employing **invariant hardware clocks**, which current major processor architectures already support [4, 6, 30, 54]. An invariant clock has a unique property: It monotonically increases at a constant frequency, regardless of dynamic frequency and voltage scaling, and resets itself to zero whenever a machine is reset (RESET signal), which is ensured by processor vendors [6, 16, 43, 54]. However, assuming that all invariant clocks in a machine are synchronized is incorrect because processors do not receive the RESET signal at the same time, so these clocks are unusable, which leads to the conclusion that we cannot compare two clocks with confidence when correctly designing any time-based concurrent algorithms. Thus, to provide a guarantee of correct use of these clocks, we propose a new primitive called ORDO, which embraces uncertainty when we compare two clock and provides an illusion of a *globally-synchronized hardware clock*. Unfortunately, accurately measuring the offset between clocks is difficult because hardware does not provide minimum bounds for the message delivery [17, 40, 47]. To solve this problem, we exploit the unique property of the invariant clock and empirically define the uncertainty window by utilizing one-way-delay latency among clocks.

Under the assumption of invariant clocks, ORDO provides an uncertainty window that remains constant while a machine is running. Thus, ORDO enables algorithms to become multicore friendly by either replacing the software clock or correctly using a timestamp with a minimal core-local computation for an ordering guarantee. We find that various timestamp-based algorithms can benefit from ORDO, which has led us to design three simple APIs. The only trick lies in handling the uncertainty window, which we explain for both physical and logical timestamp-based algorithms such as the concurrent data structure library and concurrency control algorithms for STM and databases. With our ORDO primitive, we improve the scalability of various algorithms and systems software (e.g., `RLU`, `OCC`, `Hekaton`, `TL2`, and process forking) by 1.4–39.7× across various architecture such as `Intel Xeon` and `Xeon Phi`, `AMD`, and `ARM`. Moreover, with a mere 10 lines of code change, the conventional `OCC` algorithm outperforms the state-of-the-art algorithm by 24% in a lightly contended case for the TPC-C workload.

In summary, this paper makes the following contributions:

- **A Primitive.** We design a scalable timestamping primitive called ORDO, which exposes a globally-synchronized hardware clock with an uncertainty window for various architectures by using invariant hardware clocks.
- **Correctness.** We design an algorithm that calculates the minimal offset along with its correctness, and a set of APIs that almost all sets of algorithms can easily use.
- **Applications.** We apply ORDO to five concurrent algorithms and evaluate their performance on various architectures up to 256 hardware threads.

The rest of the paper provides a background (§2), discusses the correctness proof of the ORDO primitive and its API (§3), its applicability to various algorithms (§4), implementation (§5), and evaluation (§6), and presents a discussion (§7), and then a conclusion (§8).

2. Background and Motivation

We first discuss the importance of timestamping with respect to the prior studies and then provide a brief background on the invariant hardware clocks provided by the hardware.

2.1 Time: An Ingredient for Ordering

Based on the notion of time, we classify prior research directions in terms of the applicability of clocks in two categories: logical clocks and physical timestamping.

Logical clocks. Logical timestamping, that is a software clock, is the prime primitive for any concurrent control mechanism in the domain of concurrent programming [5, 7, 8, 26, 29, 42, 44, 58] or database transactions [36, 38, 62, 66, 67]. To achieve ordering, these applications rely on atomic instructions. In the field of concurrent programming, software transactional memory [60] is an active area of research in which almost every new approach [29] is derived from time-based approaches [23, 56] that use global clocks for simplifying the validation step. To mitigate the contention on the global clock, prior studies have applied various forms of heuristics [7, 8] with several side effects in the form of false aborts, or slowly increasing timers [57], or even assumed that they have access to the synchronized clocks [58] to remove the contention from the logical clock. The use of ORDO for `STM` is inspired from prior studies [57, 58], but ORDO assumes that it has access to only fast invariant timestamp counters with an uncertainty window, which ORDO exposes for us to handle in the case of `TL2`. Similar to the `STM`, `RLU` [44] is a lightweight synchronization mechanism that simplifies the programming difficulty of RCU [46]. It also employs a global clock that serializes writers and readers, and uses a defer-based approach to mitigating the contention. We show that even with the defer-based approach, `RLU` suffers from the contention, which we address by using our ORDO primitive.

Databases also heavily rely on timestamp for concurrency control [11, 12, 36, 38, 62] to ensure serializable execution schedule. Among the various concurrency control mechanisms, the optimistic approach is popular in practice [38, 62]. Xiangyao *et al.* [66] evaluated the performance of various major concurrency control mechanisms. They emphasized

that timestamp-based concurrency control mechanisms such as optimistic concurrency control (OCC) and multi-version concurrency control mechanisms (MVCC) mostly suffer from timestamp allocation with an increasing number of cores. With the concern towards logical timestamp assignment, various OCC-based systems have explored various alternatives. For example, Silo adopted a coarse-grain timestamping at an epoch resolution and a lightweight OCC for transaction coordination within an epoch. The lightweight OCC is a conservative form of the original OCC protocol. Even though it achieves scalability, it loses its performance due to the limited CC in highly-contentious workloads [34]. In the case of MVCC protocols, they still rely on logical clocks for timestamp allocation and suffer performance collapse even in the read-only workload [66, 67]. Thus, scalable ordering primitive can be a great rescue for a large number of timestamp-based CC schemes. Also we believe that it could inspire existing systems to revisit their design decisions that were made to avoid the logical timestamp maintenance.

Physical timestamping. As multicore machines have become pervasive, physical timestamp-based algorithms are gaining more traction. For example, Oplog, an update-heavy data structure library, removes the contention from the global data structure by maintaining a per-core log that appends operations in temporal order [9]. Similarly, Dodd *et al*. [24] proposed a state-of-the-art concurrent stack that scales efficiently with the RDTSC counter. Other approaches such as quiescence mechanisms have shown that clock-based reclamation [63] eschews the overhead of the epoch-based reclamation scheme. Unfortunately, the primary concern with these algorithms is that they have access to synchronized clocks, which is not true with any available hardware. On the contrary, all clocks have constant frequency and are monotonically increasing. However, ORDO provides a globally-synchronized clock on a modern commodity processor with some uncertainty window. The only tweak that these algorithms require is a modification that ensures that they have to wait for every uncertainty window before allocating a new timestamp.

2.2 A Primer on Hardware Clocks

A physical clock, when used to mark operations, provides an externally consistent view of the operations either in a distributed or a multi-threaded environment. However, it is difficult to say the same when we obtain the ordering from two clocks, unless they are exactly synchronized which is practically impossible [2, 20, 40]. From the perspective of distributed systems, there are various clock synchronization protocols [2, 25, 40, 51, 61] that periodically synchronize the clocks either via an external clock source [47] or the clocks internally synchronize among themselves [17, 40].

We can observe a similar trend in the case of multicore machines, that now provide a per-core or per-processor hardware clocks [4, 6, 30, 54]. For example, all modern processor vendors such as Intel and AMD provide RDTSC counter, while Sparc and ARM have stick and cntvct counter, respectively. These hardware clocks are **invariant** in nature, that is the processor vendors guarantee that these clocks are monotonically increasing at a constant frequency,[2] regardless of the processor speed and its fluctuation, and reset to zero whenever a processor receives the RESET signal (reboots). To provide such invariant property, the current hardware always synchronizes these clocks from an external clock on the motherboard [15, 16, 43]. It is required because vendors guarantee that the clocks do not fluctuate even if the processor goes to the deep sleep states, which is ensured by an always running motherboard clock that runs at a constant frequency. However, the vendors do no guarantee anything regarding the synchronization of the clocks across the processor (socket) boundary for a multicore machine [30], not even within a processor because there is not guarantee regarding the delivery of the RESET signal at the same instant to all the cores inside or across the socket. We observe a similar trend in the case of an Intel and ARM machine (refer Figure 7 (a), (d)), in which one of the sockets has almost 4–8× higher measured offset than from the other direction. On the contrary, we only take the assumption that the clocks are **invariant**, that is the clocks are moving at a constant frequency and have a bounded skew once the machine is switched on. Another important point is that the synchronization of these hardware clocks via the motherboard clock do not provide the notion of the real-time. Moreover, we cannot use clock synchronization protocols in this scenario because the hardware vendors do not provide any information regarding the message delivery between cores in and across the socket boundary to provide a correct bound on the measured offset.

3. ORDO: A Scalable Ordering Primitive

ORDO relies on invariant hardware clocks to provide an illusion of a globally-synchronized hardware clock with some uncertainty. To provide such illusion, ORDO exposes three simple APIs that any application can use to figure out the time or ordering. Hence, we introduce an approach to measure the uncertainty window, along with a proof to ensure its correctness.

3.1 Embracing Uncertainty in Clock: ORDO API

For a concurrent application, these invariant clock counters are usable only if we can define the uncertainty period. Moreover, a multi-threaded program can execute on two or more threads, which requires establishing a global-clock. The global clock will not only provide a notion of monotonically increasing timestamp, but will have an uncertainty window, called ORDO_BOUNDARY, in which these hardware clocks cannot compare the order. Thus, on this basis, we provide three simple APIs (Figure 2) which the concurrent algorithms and data structures can use:

[2] These clocks may increase at a different frequency than that of the processor.

```
def get_time(): # Get current timestamp
  return hardware_timestamp()  # Timestamp instruction

def new_time(time_t t): # New timestamp after ORDO_BOUNDARY
  diff = ORDO_BOUNDARY - (t % ORDO_BOUNDARY)
  while ((new_t = get_time()) - t < diff)
    continue # pause for a while and retry
  return new_t

def cmp_time(time_t t1, time_t t2): # Compare two timestamps
  if t1 >= t2 + ORDO_BOUNDARY:   # t1 > t2
    return 1
  elif t1 + ORDO_BOUNDARY <= t2: # t2 < t1
    return -1
  return 0                       # Uncertain
```

Figure 2: ORDO clock API.

- `get_time()` is the hardware specific timestamping instruction, which also takes care of the hardware specific requirements such as instruction reordering.
- `new_time(time_t t)` returns a new timestamp that is at the `ORDO_BOUNDARY` granularity and is greater than `t`.
- `cmp_time(time_t t1, time_t t2)` compares two timestamps and establishes the precedence relation between the two based on the `ORDO_BOUNDARY`. If any of the timestamps fall within `ORDO_BOUNDARY`, then it return 0 meaning that we are uncertain.

3.2 Resolving Uncertainty in Synchronization: Using ORDO API

With ORDO APIs, another important aspect is *how to handle the uncertainty exposed by the* ORDO *primitive*. We classify the algorithms into the following categories:

- *Hardware timestamping:* Algorithms that use physical timestamping [9, 24] can use `new_time()` API to access a globally-synchronized clock to ensure their correctness.
- *Physical to logical timestamping:* Algorithms that rely on logical timestamping or versioning will benefit from all three APIs, but the most important API is the `cmp_time()`, which provides the ordering guarantee between the value of clocks. By introducing the `ORDO_BOUNDARY`, we now need to extend these algorithms to either defer or wait to execute any further operations.

3.3 Measuring Uncertainty between Clocks: Calculating ORDO_BOUNDARY

Under the assumption of invariant clocks, the uncertainty window between the clocks is constant because both clocks monotonically increase at the same frequency but may receive the RESET signal at different times. We define this uncertainty window as a physical offset (Δ). A common approach to measure Δ is to use a clock synchronization mechanism, in which a clock reads its value and the value of other clock, computes an offset and then adjusts its clock by the measured offset [21]. However, this measurement introduces various errors such as reading of the remote clocks and software overheads including the network jitter. Thus, we cannot rely on this method because 1) hardware vendors do not provide minimum bounds on the message delivery; 2) the algorithm is erroneous because of the uncertainty of the message delivery [17, 40, 47]; and 3) periodically using the clock synchronization will waste CPU cycles inside a machine, which will increase with increasing core count. Moreover, the measured offset can be either bigger or smaller than Δ, which makes it unusable for concurrent algorithms.

```
runs = 1000
shared_cacheline = {"clock": 0, "phase": INIT}

def remote_worker():
  for i in range(runs):
    while shared_cacheline["phase"] != READY:
      read_fence() # flush load buffer
    ATOMIC_WRITE(shared_cacheline["clock"], get_time())
    barrier_wait() # synchronize with the local_worker

def local_worker():
  min_offset = INFINITY
  for i in range(runs):
    shared_cacheline["clock"] = 0
    shared_cacheline["phase"] = READY
    while shared_cacheline["clock"] == 0:
      read_fence() # flush load buffer
    min_offset = min(min_offset, get_time() -
                     shared_cacheline["clock"])
    barrier_wait() # syncrhonize and restart the process
  return min_offset

def clock_offset(c0, c1):
  run_on_core(remote_worker, c1)
  return run_on_core(local_worker, c0)

def get_ordo_boundary(num_cpus):
  global_offset = 0
  for c0, c1 in combinations([0 ... num_cpus], 2):
    global_offset = max(global_offset,
                        clock_offset(c0, c1),
                        clock_offset(c1, c0))
  return global_offset
```

Figure 3: Algorithm to calculate the `ORDO_BOUNDARY`: a system-wide global offset.

3.3.1 Measuring Global Offset

To enable concurrent algorithms to correctly use our ORDO primitive, we do not synchronize these unsynchronized invariant clocks. On the contrary, we exploit the unique property of these clocks and empirically calculate a system-wide global offset, called `ORDO_BOUNDARY`, which ensures that the measured offset is always greater than the physical offset. We define a *measured offset* (δ_{ij}) as a one-way-delay latency between clocks (c_i to c_j). Suppose that the Δ between clocks is non-negative, then the measured offset in the form of one-way-delay latency, denoted as δ, will always be greater than the physical offset because of the extra one-way-delay latency. Figure 3 illustrates our algorithm to calculate the global offset after calculating the correct pair-wise offset for all the clocks in a machine. We measure the offset for core c_i and c_j as follows: c_i atomically writes its timestamp value to the variable (line 8), which notifies c_j on which it is waiting (line 16). On receiving the notification, c_j reads its own timestamp (line 18) and then calculates the difference (line 19), called δ_{ij}, and returns the value. The measured offset has an extra error over the physical offset because δ_{ij}, between cores c_i and c_j, also includes software overhead and the coherence traffic. To reduce the error, we calculate the offset multiple times and take the minimum of all the runs (line 19 – 21). To

define the correct offset between c_i and c_j, we calculate the offset from both ends (i.e., δ_{ij} and δ_{ji}) and choose the maximum of them, since we do not know which clock is ahead of the other (line 31). After calculating the offset for each core in a pair-wise fashion, we select the maximum offset as the global offset, or `ORDO_BOUNDARY`, (line 30) of that machine. This ensures that any arbitrary core is guaranteed to see a new timestamp once the `ORDO_BOUNDARY` window is over, which enables us to relate between timestamps with confidence. Moreover, the calculated `ORDO_BOUNDARY` is reasonably small because we use cache-coherence as our message delivery medium, which is the fastest source of networking between cores.

3.3.2 Correctness of Using Measured Offset

We first state our assumptions and prove with a set of lemmas that the above defined algorithm for calculating the global offset is correct with respect to invariant clocks.

Assumptions. Our primary assumption is that the clocks are invariant and they have a constant physical offset until the machine is reset. Other assumptions are that the machine follows a fail-stop model and the message delivery is reliable. The external clock synchronization protocol already takes care of the skew in the hardware clock, since the hardware guarantees an invariant clock [15, 16, 43], which is a reasonable assumption for modern, mature many-core processors such as X86, `Sparc` and `ARM`.

Lemma 1. *With invariant timestamp property, the clock offset remains constant.*

Proof. An invariant clock c, on a core i, runs with a fixed monotonic frequency and returns a local clock time $c_i(t)$ at a real time t. Hence, a clock updates its counter at every clock tick and is a discrete function with a constant increasing value:

$$c_i(t) = C(t) + \Delta_i$$
$$\Delta_{ij} = c_i(t) - c_j(t) = \Delta_i - \Delta_j \quad (1)$$

where $C(t)$ is the invariant clock, which synchronizes with an external hardware clock to ensure that its invariant property provided by the hardware; Δ_i and Δ_j are the physical offset of clocks c_i and c_j, which is present because of the delay in receiving the RESET signal at the time of boot. The clocks have the invariant property, i.e., Δ_i and Δ_j are constant regardless of dynamic frequency and voltage scaling (DVFS), and deep sleep state [15, 16, 43, 54]. Therefore, Δ_{ij} obtained in Equation 1 is also constant, i.e., the offset between two clocks will remain constant while the machine is running. □

These clocks are usable for concurrent algorithms, only if we can calculate the Δ_{ij}. However, calculating Δ_{ij} is hardware dependent and it is even difficult for the hardware vendor to provide such information since it will be dependent on the various configurations that include motherboards and processors. To overcome this issue, we need to prove that the correct measured offset ($\delta_{i\leftrightarrow j}$) is the maximum of the measured offsets of clocks in both directions, and the global offset will be the maximum of all the pair-wise calculated offsets, which leads to another lemma.

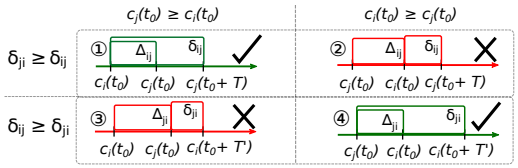

Figure 4: Calculating the offset ($\delta_{i\leftrightarrow j}$) by using the pair-wise one-way-delay latency between the clocks (c_i and c_j). Δ_{ij} and Δ_{ji} is the physical offset between c_i and c_j, and c_j and c_i respectively. δ_{ij} and δ_{ji} are the measured offset by using our approach. There are four possible cases depending on the physical offset and the measured value. The key point is that we consider each direction separately for measuring the offset, instead of averaging out the calculated latency, which breaks the invariant that the measured offset is always greater than the physical offset.

Lemma 2. *The maximum of the calculated offsets from c_i to c_j and c_j to c_i pairs (i.e., $\delta_{i\leftrightarrow j} = \max(\delta_{ij}, \delta_{ji})$) is always greater than or equal to the physical offset (i.e., Δ_{ij}).*

Proof. For cores c_i and c_j:

$$\delta_{ij} = |c_j(t_1) - c_i(t_0)| \quad = |c_j(t_0 + T) - c_i(t_0)| \quad (2)$$

$$\delta_{ji} = |c_i(t_1) - c_j(t_0)| \quad = |c_i(t_0 + T') - c_j(t_0)| \quad (3)$$

δ_{ij} and δ_{ji} denote the offset measured from c_i to c_j and c_j to c_i respectively and $|\Delta_{ij}| = |\Delta_{ji}|$ from lemma 1; $c_i(t_0)$ and $c_j(t_0)$ denote the clock value at a real time t_0, which is a constant monotonically increasing function with a defined timestamp frequency by the hardware. T and T' are the recorded true time when they receive the message from another core (line 16), which denotes the one-way-delay latency in either direction (line 31). As there are two physical offset possible (Δ_{ij} and Δ_{ji}), there are four possible values that we may measure depending on the one-way-delay latency. Figure 4 illustrates four cases that are possible from Equation 2 and Equation 3. If $c_i(t) \leq c_j(t)$, case ① and ③ are the possible outcomes of δ_{ij} and δ_{ji} respectively, which ensures that case ① has a larger offset (δ_{ij}), else case ④ has larger offset (δ_{ji}) if $c_i(t) \geq c_j(t)$. Thus, either ① or ④ ensure that the measured offset is always greater than the physical offset, but only one of them holds at a time because the clocks are monotonically increasing at constant frequency. Hence, if we take the maximum of the measured offsets for the permutation of c_i and c_j, then one of the cases ① or ④ always hold true and guarantee that the measured offset is actually greater than the physical offset. There is a possibility that cases ② or ③ give a higher offset because of the higher overhead in measuring the offset (δ), still our invariant holds correct as the measured offset is greater than the ones measured by ① and ④ which ensure that the value those cases is always bigger than the physical offset. □

Theorem. *The global offset is the maximum of all the pair-wise measured offsets for each core.*

Proof.

$$\delta^g = \max(\delta_{i \leftrightarrow j}) \quad = \max(\max(\delta_{ij}, \delta_{ji})) \mid \forall i, j \in \{0..N\}$$

N is the number of cores and δ^g is the `ORDO_BOUNDARY`. We extend lemma 2 for all the pair-wise combinations of cores (refer Figure 3 line 29) to calculate a vector of pair-wise maximum offsets among all the cores and take the maximum of all of them because: 1) it ensures that any arbitrary clock always reads a new timestamp when one of the timestamp window as long as the `ORDO_BOUNDARY` is over; and 2) it also acts as an uncertainty period in which we cannot definitively provide the precedence relationship between two timestamps if both of them fall in the same window. □

4. Use cases

We now demonstrate the application of the ORDO primitive on five concurrent algorithms that either employ physical or logical timestamping. The key question, we address, is how to handle the uncertainty window (`ORDO_BOUNDARY`) to compare two timestamps without breaking the correctness of these algorithms.

4.1 `Oplog`: An Update-heavy Data Structures Library

`Oplog` [9] is a per-core log-based library that scales update-heavy data structures. It maintains a per-core log, which stores the update operations in a temporal order. During the read phase, it applies the logs on the data structure in a centralized, time-order manner. To ensure the correct temporal ordering across all per-core logs, `Oplog` requires a system-wide synchronized clock to determine the ordering of the operations. We modify `Oplog` to use the `new_time()` API that has a notion of a globally-synchronized hardware clock when appending the operations to a per-core log and use the `cmp_time()` to compare timestamps. `Oplog` ensures linearizability by relying on the system-wide synchronized clock, which ensures that the obtained timestamps are always in increasing order. Our modification guarantees linearizability because `new_time()` exposes a globally synchronized clock, which always provides a monotonically increasing timestamp that will always be greater than the previous value that is already passed across all the invariant clocks. There is a possibility that two or more timestamps fall inside the `ORDO_BOUNDARY` at the time of merge phase, which denotes concurrent update operations, and it is also possible in the original `Oplog` design. We use similar techniques used by `Oplog` to apply these operations in an ascending order of the core id.

4.2 Read-Log-Update (`RLU`)

`RLU` is an extension to the RCU framework that enables a semi-automated method of concurrent read-only traversals with multiple updates. `RLU` follows the design of an STM algorithm, and it maintains an object-level write-log per thread, in which the writers first lock the object, then copy them in their own write log and manipulate them locally without affecting the original structure, and adopts the RCU barrier mechanism with a global clock-based logging mechanism [8, 23].

```
1    # All operations acquire the lock
2    def rlu_reader_lock(ctx):
3      ctx.is_writer = False
4      ctx.run_count = ctx.run_count + 1 # Set active
5      memory_fence
6  -   ctx.local_clock = global_clock # Record global clock
7  +   ctx.local_clock = get_time() # Record ORDO global clock
8
9    def rlu_reader_unlock(ctx):
10     ctx.run_count = ctx.run_count + 1 # Set inactive
11     if ctx.is_writer is True:
12       rlu_commit_write_log(ctx) # Write updates
13 ------------------------------------------------------------
14   # Pointer derefernce
15   def rlu_dereference(ctx, obj):
16     ptr_copy = get_copy(obj) # Get pointer copy
17     if is_unlocked(ptr_copy): # Is obj free?
18       return obj # return current obj
19     if is_copy(ptr_copy): # Already a copy?
20       return obj # return obj
21     thread_id = get_thread_id(ptr_copy)
22     if thread_id == ctx.thread_id: # Locked by current ctx
23       return ptr_copy # Return copy
24     other_ctx = get_ctx(thread_id) # check for stealing
25 -   if other_ctx.write_clock <= ctx.local_clock:
26 +   if cmp_time(ctx.local_clock, other_ctx.write_clock) == 1:
27       return ptr_copy # return ptr_copy (stolen)
28     return obj # no stealing, return the oject
29 ------------------------------------------------------------
30   # Memory commit
31   def rlu_commit_write_log(ctx):
32 -   ctx.write_clock = global_clock + 1 # Enable stealing
33 -   fetch_and_add(global_clock, 1) # Advance clock
34 +   ctx.write_clock = new_time(ctx.local_clock) # Get ORDO clock
35     rlu_synchronize(ctx) # Drain readers
36     rlu_writeback_write_log(ctx) # Safe to write back
37     rlu_unlock_write_log(ctx)
38     ctx.write_clock = INFINITY # Disable stealing
39     rlu_swap_write_logs(ctx) # Quiesce write logs
40 ------------------------------------------------------------
41   # Synchronize with all threads
42   def rlu_synchronize(ctx):
43     for thread_id in active_threads:
44       other = get_ctx(thread_id)
45       ctx.sync_cnts[thread_id] = other.run_cnt
46     for thread_id in active_threads:
47       while:
48         if ctx.sync_cnts[thread_id] & 0x1 != 0:
49           break # not active
50         other = get_ctx(thread_id)
51         if ctx.sync_cnts[thread_id] != other.run_cnt:
52           break # already progressed
53 -       if ctx.writer_clock <= other.local_clock:
54 +       if cmp_time(other.local_clock, ctx.writer_clock) == 1:
55           break # started afer me
```

Figure 5: `RLU` pseudo code including our changes.

Figure 5 illustrates the pseudo-code for some of the main functions of `RLU` that use the global clock. We refer to the pseudo-code to explain its working, limitations, and our changes to the `RLU` design. `RLU` works as follows: all operations reference the global clock in the beginning (line 2) and rely on it to dereference the shared objects (line 15). In the case of write operation, each writer first dereferences the object and copies it in its own log after locking the object. At the time of commit (line 31), it increments the global clock (line 33), which effectively splits the memory snapshots into the old and new one. While old readers refer the old snapshot who have smaller clock values than the incremented global clock, the new readers refer to the new snapshot that starts after the increment of the global clock (lines 24 – 28). Later,

after increasing the global clock, writers wait for the old readers to finish by executing the RCU style quiescence loop (lines 46 – 55), while the new operations obtain the new objects from the writer log. As soon as the quiescence period is over, the writer safely writes back the new objects from its own log to the shared memory (line 36) and then releases the lock (line 37). In summary, RLU has three scalability bottlenecks: 1) maintaining a global clock and referencing it, which will not scale with an increasing core count (Figure 1); 2) the object lock for manipulation, and 3) copying the object for the write operation. RLU tries to mitigate the first problem by employing a defer-based approach, but it comes at the cost of extra memory utilization. The last two are the design choices that prefer programmability over the hand-crafted copy management mechanism.

We remove the scalability bottleneck in updating the global clock by using the ORDO primitive. For each operation, all operations reference the global clock via `get_time()` (line 6), and rely on it to dereference the object by comparing the timestamp for the two contexts with the `cmp_time()` API (line 7), which provides the same comparison semantics as the earlier comparison (line 6). In the case of writers, we demarcate between the old and new snapshot by obtaining a new timestamp via the `new_time()` API (line 34), and later rely on this timestamp for the clock-based quiescence period for the readers that are using the old snapshot (line 54).

Our modification does not break the correctness of the RLU algorithm, which is to always have a consistent memory snapshot in the RLU protected section. In other words, a protected RLU section, at time t, will not see its concurrent overwrite that will happen at time $t' > t$. Since, the invariant hardware clocks provide a constantly monotonically increasing clock, the time obtained at the commit time of the writer (line 34) is always greater than the previous write timestamp. Our modification enforces the variant for the writers in the following ways. 1) If a new writer, is able to steal the copy from the other writer (line 26), then it still holds the invariant. 2) If a new writer is unable to steal the copy of the locked object (line 28), then the old writer will quiesce while holding the writer lock, (line 42), which will force the new writer to abort and retry because RLU does not allow writer-writer conflict. Note that the RLU algorithm already takes care of the writer log quiescence by maintaining two version of the logs, which are swapped to allow the stealing readers to use them (line 39).

4.3 Concurrency Control for Databases

Database systems serve highly concurrent transactions and queries, with strong consistency guarantees by using a concurrency control (CC) scheme. There are two main CC schemes that are popular among the state-of-the-art database systems: optimistic concurrency control (OCC) and multi-version concurrency control (MVCC). Both schemes rely on a timestamp to decide global commit order and to avoid significant overhead of lock management [32].

OCC works by breaking a transaction execution into three phases: 1) *read*, 2) *validation* and 3) *write*. In the first phase, a worker keeps footprints of the transaction in local memory. At commit time, it acquires locks on the write set in a certain order to prevent lost update and deadlock. After that, it moves on to *validation* phase to check whether the transaction violates the serializability. It assigns a global commit timestamp to the transaction and validates both the read and write set by comparing their timestamps with the commit timestamp to ensure serializability. After passing the *validation* phase, the worker in a transaction enters *write* phase in which it updates its write set visible by overwriting original tuples and releasing held locks. To address the problem of logical timestamps in OCC [36], some state-of-the-art OCC schemes have mitigated the updates either with conservative read validation [35, 62] or data-driven timestamping [67].[3] To show the impact of ORDO, we modify the first two phases of the OCC algorithm [36] (read and validation phase), denoted as OCC$^{\text{ORDO}}$. In the *read phase*, we assign the timestamp via the `get_time()` API, which guarantees that the new timestamp will be greater than the previous one. The validation scheme is same as before, but the only difference is that the `get_time()` provides the commit timestamp. The worker then uses the commit timestamp to validate the read set by comparing it with the recorded timestamp of the read and write set. We apply a conservative approach of aborting the transactions if two timestamps fall within the `ORDO_BOUNDARY` in the validation step. There are two requirements for OCC$^{\text{ORDO}}$ to ensure serializability: 1) obtain a new timestamp which `new_time()` ensures; and 2) handling the uncertainty window, in which we conservatively abort the transactions, which also resolves the later-conflict check [3] to ensure the global ordering of the transactions.

MVCC is another major category of CC schemes. Unlike other single-version CC schemes, it takes an append-only update approach to spare multiple read copies. Then, a reader locates a desired version by performing "time traveling" on the version history. MVCC avoids reader-writer conflicts by forwarding them to physically different places, thus it is more robust under high contention than OCC. To support the time traveling and deciding the commit order, MVCC also relies on logical timestamping, thereby suffering from the timestamp allocation [66]. ORDO can be a drop-in replacement for the software timestamp, providing superior multicore scalability while preserving the inherent robustness of MVCC against contention. To demonstrate how an existing MVCC system can be accelerated through ORDO, we choose Hekaton, a state-of-the-art in-memory database system [38]. Although it supports multiple CC levels with varying consistency guarantee, we focus on serializable, optimistic MVCC mode.

[3] Lazy timestamping is similar to stutter-based counters [37] in which a transaction is assigned a local timestamp and increments by one more among the maximum of all local timestamp values.

We first describe how the original algorithm works and moves on the modifications to introduce ORDO into the system.

A transaction starts from a begin timestamp allocation from the global clock to decide which version to read for its lifetime. During the *read* stage, a transaction is allowed to read a version when its begin timestamp falls between the begin and end timestamps associated with the version. For update, a new version is appended to database immediately, with the transaction ID (TID) of the owner transaction marked in the begin timestamp field. At commit, a transaction is assigned a commit timestamp to determine the serialization order, validates read/scan set and iterates write set to replace TIDs installed in the begin timestamp field with the commit timestamp. A transaction encountering a TID-installed version examines visibility with the begin/end timestamps of the owner transaction instead. The first modification required is to replace the allocation with the `get_time()` API to obtain a new timestamp from the invariant clock. Also, we need to remove atomic instructions used for logical clock maintenance to entirely eliminate its overhead.

As a result, this modification introduces uncertainty to compare timestamps from different local clocks during the visibility check. We substitute the comparison with `cmp_time()` to ensure correctness. Comparing timestamps is allowed only with larger time difference than `ORDO_BOUNDARY` because it provides a definite precedence relation. Otherwise, the comparison fails and a transaction is forced to start over or abort due to the uncertainty. However, given the small size of `ORDO_BOUNDARY`, we expect the aborts caused by uncertainty to be rare. In terms of correctness, our system provides same consistency guarantee as the original one, but it is more rigorous because it aborts more transactions due to `cmp_time()` failure by uncertainty.

4.4 Software Transactional Memory (TL2)

We choose TL2, an ownership- and word-based STM algorithm that employs timestamping for reducing the common-case overhead of validation. TL2 works by ordering the update and access relative to the global logical clock and only checks the ownership record once, i.e., it validates all the transactional reads at the time of commit. The algorithm works as follows: 1) a transaction begins by storing a global time value ($start$). 2) For a transactional load, it first checks whether the orec [4] is unlocked and has no newer timestamp than the start timestamp of the transaction, and then adds the orec to its read set, and appends the address-value pair in the case of a transactional write. 3) At the time of commit, it first acquires all the locks in the write set, and then validates all the elements of the read set by comparing the timestamp of the orec with that of the start timestamp of the transaction. If successful, the transaction writes back the data and obtains a new timestamp which denotes the end of the timestamp (end). It uses the end timestamp as a linearizable point which it atomically writes in each orec of the write set (write address) and also releases the lock. Here, end is guaranteed to be greater than the $start$ timestamp as the transaction atomically increments a logical global clock, which in turn makes each transaction update as linearizable.

The requirement of the TL2 algorithm is that the end timestamp should be greater than the $start$ value, which the ORDO primitive always provide; moreover, two disjoint transactions can share the same timestamp [69]. Thus, by using the ORDO APIs, we modify the algorithm as follows: we modify $start$ to use the `get_time()` API to obtain a time stamp from the invariant clocks and use `new_time`($start$) to obtain a definite newer timestamp for the end variable. Here, we again adopt a conservative approach of aborting the transactions if the two timestamps fall in the `ORDO_BOUNDARY`, as this can corrupt the memory or result in an undefined behavior of the program [23]. We can use timestamp extension to remove the aborts that occur in the read timestamp validation at the beginning of a transactional read, although it may not benefit us because of the small `ORDO_BOUNDARY`. Our modification ensures linearizability by 1) first providing an increasing timestamp via `new_time()` API which globally provides a new timestamp value; and 2) by always validating the read set at the time of commit. Again we abort the transactions if the two timestamps (read set timestamp and commit timestamp) fall in the `ORDO_BOUNDARY`, we abort and retry the transactions to remove the uncertainty during the validation phase. Similarly, while performing a transactional load, we apply the same strategy to remove any inconsistencies.

5. Implementation

Our library and micro benchmarks comprise 200 and 1,100 lines of code (LoC) respectively, which support architecture specific timers for different architectures. We modify various programs to show the effectiveness of our ORDO primitive. We modify 50 LoC in the `RLU` code base to support both ORDO primitive and the architecture specific code for `ARM`. To specifically evaluate database concurrency protocols, we choose DBx1000 [65] because it includes all of the database concurrency control algorithms. We modify 400 LoC to support both `OCC`$^{\texttt{ORDO}}$ and `Hekaton`$^{\texttt{ORDO}}$ algorithms, including changes for the `ARM` architecture. We use an x86 port of the TL2 algorithm [49] that we extend (25 LoC) using the API of the ORDO primitive.

6. Evaluation

We evaluate the effectiveness of the ORDO primitive by addressing the following key questions:

- How does an invariant hardware clock scale on various commodity architectures? (§6.1)
- What is scalability characteristic of the ORDO primitive? (§6.2)

[4] Orec is a ownership record, which either stores the identity of a lock holder or the most recent unlocked timestamp.

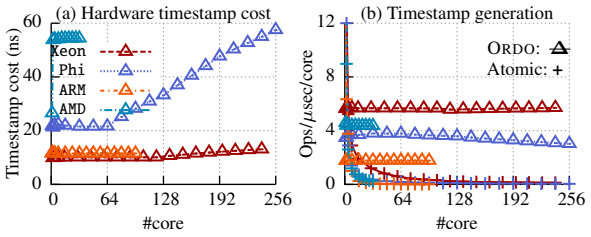

Figure 6: Micro evaluation of the invariant hardware clocks used by the ORDO primitive. (a) show the cost of a single timestamping instruction when it is executed by varying the number of threads in parallel. (b) shows the number of per-core generated timestamps in a micro second with atomic increments (+) and with `new_time()` API (▲) that generates timestamps after every `ORDO_BOUNDARY` window.

Machine	Cores	SMT	Speed (GHz)	Sockets	ORDO_BOUNDARY min (ns)	ORDO_BOUNDARY max (ns)
Intel Xeon	120	2	2.4	8	70	276
Intel Xeon Phi	64	4	1.3	1	90	270
AMD	32	1	2.8	8	93	203
ARM	96	1	2.0	2	100	1,100

Table 1: Various machine configuration that we use in our evaluation as well as the calculated `ORDO_BOUNDARY` which we use in our evaluation. While min is minimum `ORDO_BOUNDARY` between cores, max is the global offset.

- What is the impact of the ORDO primitive on algorithms that rely on synchronized clocks? (§6.3)
- What is impact of the ORDO primitive on version-based algorithms? (§6.4, §6.5, §6.6)
- How does the `ORDO_BOUNDARY` affect the scalability of algorithms? (§6.7)

Experimental setup. Table 1 lists the specifications of four machines, namely, a 120-core Intel `Xeon` (having two hyperthreads), a 64-core Intel `Xeon Phi` (having four hyperthreads), a 32-core `AMD`, and a 96-core `ARM` machine. The first three machines have x86 architecture, out of which `Xeon` has higher number of physical cores and sockets, `Phi` has higher degree of parallelism, and `AMD` is from a different processor vendor. These three processors specify invariant hardware clocks in their specification. We also use a 96-core `ARM` machine for our evaluation, whose clock is different from existing architectures. It supports a separate generic timer interface, which exists as a separate entity inside a processor [6].

6.1 Scalability of Invariant Hardware Clocks

We create a simple benchmark to measure the cost of hardware timestamping instruction on various architectures. The benchmark forks a process on each core and repeatedly issues the timestamp instruction for a period of 10 seconds. Figure 6 shows the cost of timestamp instruction on four different architectures. We observe that the cost of timestamp operations remains constant up to the physical core count, but increases with increasing hardware threads which is evident in `Xeon` and `Phi`, but it is still comparable to an atomic instruction operation in a medium contended case. One important point is that `ARM` supports a scalable timer whose cost (11.5 ns) is equivalent to that of `Xeon` (10.3 ns). In summary, the current hardware clocks are a suitable foundation for mitigating the contention problem of global clock with increasing core count, potentially together with hyperthreads.

6.2 Evaluating ORDO Primitive

Figure 7 presents the measured offset (δ_{ij}) for each pairwise combination of the cores. The heatmap shows that the measured offset between two adjacent clocks inside a socket is the least on every architecture. One important point is that all the measured offsets are positive. As of this writing, we never encountered a single negative measured offset while measuring the offset in either direction from any core over the course of past two months. This holds true with prior results [9, 58, 63], and illustrates that the added one-way-delay latency is greater than the physical offset which always results in giving a positive measured offset in any direction. For a certain set of sockets, we observe that the measured offset, within that socket, is less than the global offset. For example, the fifth socket in `Xeon` machine has a maximum offset of 120 ns compared with the global offset of 276 ns.

We can define the `ORDO_BOUNDARY` based on the application use in which application can choose the maximum offset of a subset of cores or sockets inside a machine. But, they need to embed the timestamp along with the core or thread id, which will shorten the length of the timestamp variable and is not advantageous (§6.7). Therefore, we choose a global offset across all the cores as the `ORDO_BOUNDARY` for all of our experiments. Table 1 shows the minimum and maximum measured offset for all of the evaluated machines with offset is the `ORDO_BOUNDARY`. We create a simple micro benchmark to show the impact of timestamp generation by each core by using our `new_time()` and the atomic increments. Figure 6 (b) shows the results of obtaining a new timestamp from a global clock, which is a representative of both physical timestamping and read-only transactions. The ORDO-based timestamp generation remains almost constant up to the maximum core count. It is 17.4–285.5× faster than the atomic increment at the highest core count, thereby showing that the transactions that use logical timestamp will definitely improve their scalability with increasing core count.

One key observation is that one of the socket in both `Xeon` (eighth socket: 105–119 cores) and `ARM` (second socket: 48–96 cores) have 4–8× higher measured offset when measured from a core belonging to the other socket, even though the measured socket bandwidth is constant for both architecture [41]. For example, the measured offset from core 50 to 0 is 1,000 ns, but only 100 ns from core 0 to 50 for the `ARM` machine. We believe that one of the sockets received the RESET signal later than the other sockets in the machine, thereby showing that the clocks are not synchronized. In the case of `Phi`, we observe that most of the offset lies in the window of 200 ns, but the adjacent cores have the least offset.

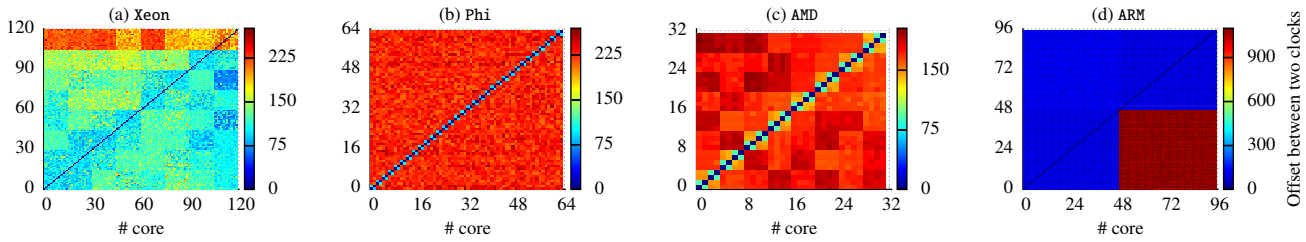

Figure 7: Clock offsets for all pairs of each cores. The measured offset varies from a minimum of 70 ns to 1,100 ns for all architectures. Both `Xeon` (a) and `ARM` (d) machines show that the one of the sockets have 4–8× higher offset than the others. To confirm this, we measured the bandwidth between the sockets, which is symmetric in both machines.

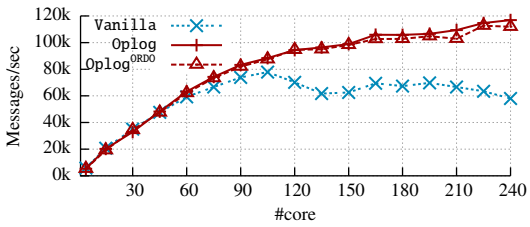

Figure 8: Throughput of `Exim` mail-server on 240-core machine. `Vanilla` represents the unmodified Linux kernel, while `Oplog` is the `rmap` data structure modified by the `Oplog` API in the Linux kernel. `Oplog`$^{\text{ORDO}}$ is the extension of `Oplog` with the ORDO primitive.

6.3 `Oplog` (Physical Timestamping)

For physical timestamping, we evaluate the impact of `Oplog` on the Linux reverse map (`rmap` [45]), which records the page table entries that map a physical page for every physical page, and it is primarily used by `fork()`, `exit()`, `mmap()`, and `mremap()` system calls. We modify the reverse mapping for both anonymous and file rmaps in the Linux kernel [9]. We use `Exim` [1] mail-server on the `Xeon` machine to evaluate the scalability of the `rmap` data structure. `Exim` is a part of the `Mosbench` [14] benchmark suite, and it is a process intensive application which listens to SMTP connections and forks a new process for each connection, and a connection forks two more processes to perform various file system operations in shared directories as well as on the files. Figure 8 shows the results of `Exim` on the stock Linux (`Vanilla`) and our modifications that include kernel versions with (`Oplog`$^{\text{ORDO}}$) and without the ORDO primitive (`Oplog`). The results show that `Oplog` does alleviate the contention from the reverse mapping, which we observe after 60 cores and the throughput of `Exim` increases by 1.9× at 240 cores. The `Oplog` version is merely 4% faster than the `Oplog`$^{\text{ORDO}}$ approach because `Exim` is now bottlenecked by the file system operations [48] and zeroing of the pages at the time of forking after 105 cores.

6.4 Read Log Update

We evaluate the throughput of `RLU` for hash table benchmark and citrus tree benchmark across architectures. The hash table uses one linked list per bucket and the key hashes into the bucket and traverses the linked list for a read or write operation. Figure 9 shows the throughput of hash table with varying update rates of 2%, 20%, and 40% across the architectures, where `RLU` is the original implementation and `RLU`$^{\text{ORDO}}$ is the modified version. The results show that `RLU`$^{\text{ORDO}}$ outperforms the original version by an average of 2.1× across all the architectures for all update ratios at the highest core. Across architectures, the result for the starting number of cores for `RLU` is better than `RLU`$^{\text{ORDO}}$ because 1) coherence traffic until certain core counts (`Xeon`: within a socket, `Phi`: until 4 cores, `ARM`: 36 cores and `AMD`: 12 cores) assists the `RLU` protocol that has lower abort rates than `RLU`$^{\text{ORDO}}$, and 2) `RLU`$^{\text{ORDO}}$ has to always check for the locks while dereferencing because the invariant clock does not provide the semantic of `fetch_and_add()`. Thus, in the case of only readers (100% reads), `RLU`$^{\text{ORDO}}$ is 8% slower than `RLU` at highest core across the architectures, but even with a 2% update rate, it is the atomic update that degrades the performance of `RLU`.

Overall, all multisocket machines show similar scalability trend after crossing the socket boundary, and are saturated after certain core count because they are bottlenecked by the locking and creation of the object and its copy, which is more severe in the case of `ARM` for crossing the NUMA boundary as evident after 48 cores. In the case of `Phi`, there is no scalability collapse because it has a higher memory bandwidth and has slower processor speed, which only saturates the throughput. However, as soon as we remove the logical clock, the throughput increases by an average of 2× for all of the cases. Even though the cost of timestamp instruction increases with hyperthreads (3× at 256 threads), the throughput is almost saturated because of the object copying and locking. Even with the deferrals (refer Figure 11), `RLU`$^{\text{ORDO}}$ is at most 1.8× faster than the `RLU` algorithm, illustrating that the defer-based approach still suffers from the contention on the global clock. We also evaluate the citrus-tree benchmark that involves complex update operations. Figure 10 shows the results of the citrus-tree benchmark for the 10%, 20%, and 40% updates across various operations. We can observe that `RLU`$^{\text{ORDO}}$ outperforms `RLU` even in the complicated scenario by a factor of two on every architecture. The scalability behavior is similar to the hash table benchmark across the architectures.

6.5 Concurrency Control Mechanism

We evaluate the impact of ORDO primitive on the existing `OCC` and `Hekaton` algorithms with YCSB [19] and TPC-C benchmark. We execute read-only transactions to only focus on scalability aspect without transaction contentions (Figure 12), which comprises two read-queries per transaction and a uni-

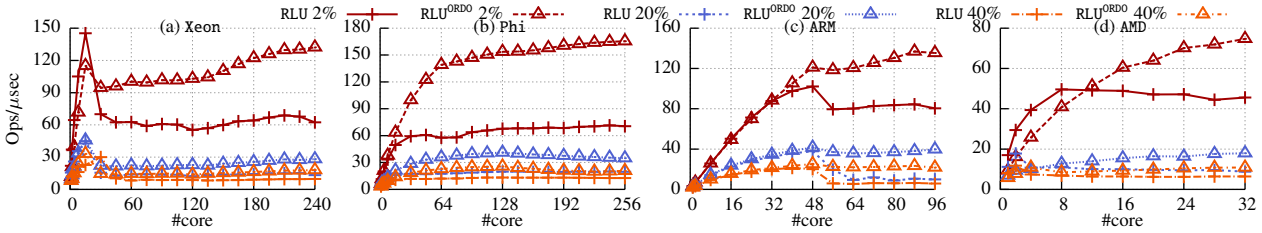

Figure 9: Throughput of the hash table with RLU and RLU^{ORDO} for various update ratios of 2%, 20%, and 40% updates. The user space hash table has 1,000 buckets with 100 nodes. We experiment it on four machines from Table 1.

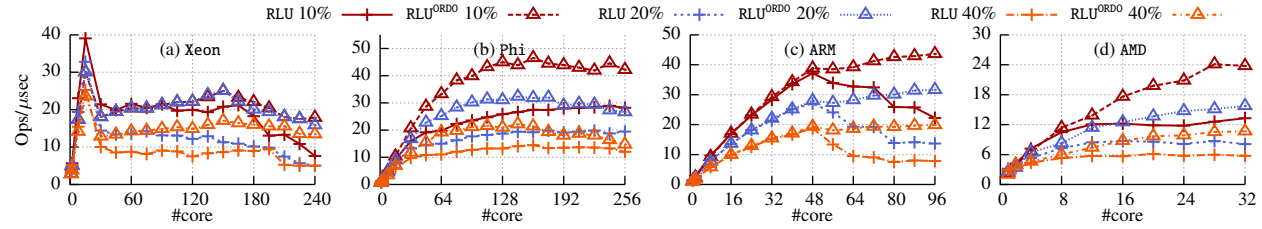

Figure 10: Throughput of the citrus tree with RLU and RLU^{ORDO} that consists of 100,000 nodes with varying updates ratio of 10%, 20%, and 40% on various machines.

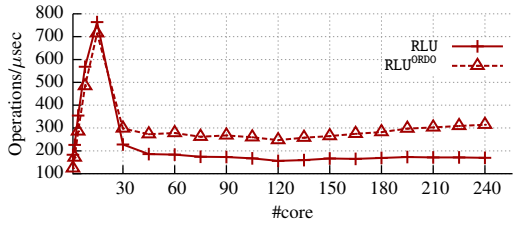

Figure 11: Throughput of the hash table benchmark (40% updates) with the deferred-based RLU and RLU^{ORDO} approach on the Xeon machine. Even with the deferred-based approach, the cost of global clock is still visible after crossing the NUMA boundary.

form random distribution. We also present results from TPC-C benchmark with 60 warehouses as a contentious case (Figure 14). Figure 12 shows that both OCC^{ORDO} and Hekaton^{ORDO} outperform OCC and Hekaton by 5.6–39.7× and 4.1–31.1× respectively. The reason for such improvement is because both OCC and Hekaton wastes 62–80% of their execution time in allocating the timestamps, whereas ORDO successfully eliminates the logical timestamping overhead with a modest amount of modification. Compared with state-of-the-art optimistic approaches that avoid global logical timestamping, OCC^{ORDO} shows comparable scalability, thereby enabling ORDO to serve as a push-button accelerator for timestamp-based applications, which provides a simpler way to scale. In addition, Hekaton^{ORDO} has comparable performance to that of other OCC based algorithms.

Figure 14 presents the throughput and abort rate for the TPC-C benchmark. We run NewOrder (50%) and Payment (50%) transactions only with hash index. The results show that OCC^{ORDO} is 1.24× faster than TicToc and has 9% less abort rate, since TicToc starts to spend more (7%) time in the validation phase because of the overhead of its data-driven timestamp computation. This proves that hardware clocks provide better scalability than software bypasses that come at the cost of extra computation. In the case of Hekaton^{ORDO}, it also outperforms Hekaton by 1.95× with less aborts.

6.6 Software Transactional Memory

We evaluate the impact of ORDO primitive by evaluating the TL2 and TL2^{ORDO} algorithms on the set of STAMP benchmarks. Figure 13 presents the speedup over the sequential execution of these benchmarks. The results show that TL2^{ORDO} improves the throughput of every benchmark except Labyrinth, which is bottlenecked by the validation and instrumentation of the loads and stores than the clock itself. TL2^{ORDO} has higher speedup over TL2 for both ssca2 and Kmeans because they have short transactions, which in turn results in more clock updates. Genome, on the other hand, is dominated by large read conflict-free transactions, thus, it does not severely stresses the global clock with an increasing core count. In the case of Intruder, we observe some performance improvement until 60 cores. However, after 60 cores, TL2^{ORDO} has 10% more aborts than TL2, which in turn slows down the performance of the TL2^{ORDO} as the bottleneck shifts to the large working set maintained by the basic TL2 algorithm. We can circumvent this problem by employing type-aware transactions [29]. Finally, TL2^{ORDO} also improves the performance of Vacation because it is a transactional-intensive workload as it performs at least 3 M transactions with abort rates in the order of 300K–400K, which results in stressing the global clock. In summary, ORDO primitive improves the throughput of the TL2 by a maximum factor of two.

6.7 Sensitivity Analysis on ORDO_BOUNDARY

To show that the calculated ORDO_BOUNDARY does not hamper the throughput of the algorithm, we ran the hash table benchmark with RLU^{ORDO} algorithm at 240 cores with varying offsets. We found that the offset boundary than lower ORDO_BOUNDARY does not affect the scalability because RLU has other bottle-

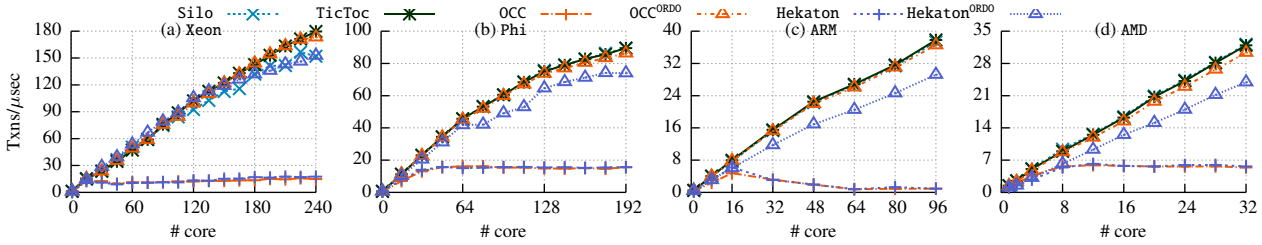

Figure 12: Throughput of various concurrency control algorithms of databases for the read-only transactions (100% reads) using the YCSB benchmark on various architectures. We modify the existing `OCC` and `Hekaton` algorithm to use our ORDO primitive (`OCC`$^{\text{ORDO}}$ and `Hekaton`$^{\text{ORDO}}$ respectively) and compare them against the state-of-the-art `OCC` algorithms: `Silo` and `TicToc`. Our modifications removes the logical clock bottleneck across various architectures.

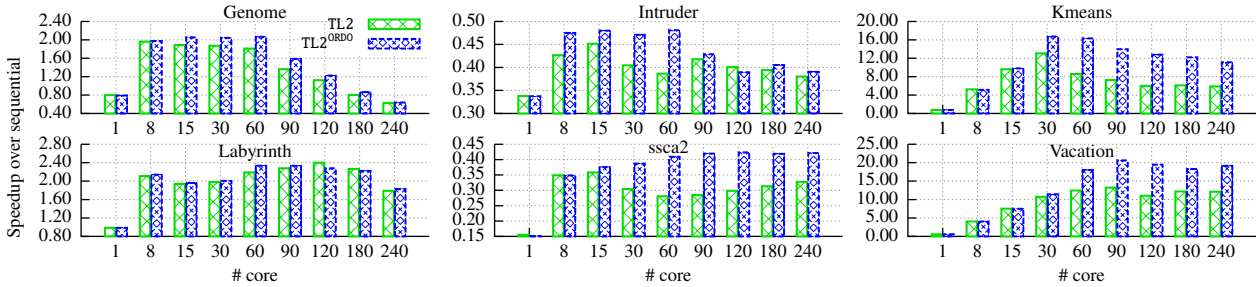

Figure 13: Speedup of STAMP benchmark with respect to the sequential execution on `Xeon` machine for `TL2` and `TL2`$^{\text{ORDO}}$ algorithms.

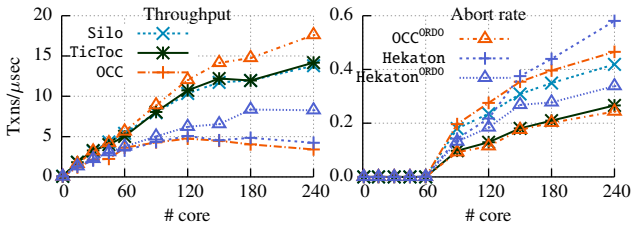

Figure 14: Throughput and abort rates of concurrency control algorithms for TPC-C benchmark with 60 warehouses on 240 `Intel Xeon` machine.

necks, which overshadow the cost of both lower and higher offsets. We can further confirm this result from Figure 6 (b) results as hardware clocks can generate 1.4 B timestamps at 240 cores, while the maximum throughput of the hash table is only 102 M for 2% update operation. We observe the similar trend on a single core as well as for multiples of sockets.

7. Discussion

ORDO enables applications or systems software to correctly use invariant hardware timestamps, however the uncertainty window can play a huge part for large multicore machines that are going to span beyond thousand cores. It is possible to further reduce the uncertainty window if the processor vendors can provide some bound on the cost of cache line access, which will allow us to use the existing clock synchronization protocols with confidence, thereby decreasing the uncertainty window as well as increasing the number of generated timestamps (refer Figure 6). Moreover, other hardware vendors should provide invariant clocks that will enable applications to easily handle the ordering in a multicore friendly manner.

The notion of a globally-synchronized hardware clock opens up a plethora of opportunities in various areas. For example, applications like Doppel [52] and write-ahead-logging [39] schemes can definitely benefit from the ORDO primitive. Since, an OS can easily measure the offset of the invariant clocks, it is not applicable to virtual machines (VM). We believe that the hypervisor should expose this information to VMs for taking full advantage of the invariant hardware clocks, which the cloud providers can easily expose via the clock ABI. ORDO can be one of the basic primitives for the upcoming rack-scale computing architecture, in which it can inherently provide the notion of bounded cache-coherence consistency for the non-cache coherent architecture.

8. Conclusion

Ordering still remains the basic building block to reason about the consistency of operations in a concurrent application. However, ordering comes at the cost of expensive atomic instructions that do not scale with increasing core count, and further limit the scalability of the concurrent on large multicore and multi-socket machines. We propose ORDO, a simple scalable primitive that addresses the problem of ordering by providing an illusion of a globally-synchronized hardware clock with some uncertainty inside a machine. ORDO relies on the invariant hardware clocks that are guaranteed to be increasing at a constant frequency, but are not guaranteed to be synchronized across the cores or sockets inside a machine, which we confirm for `Intel` and `ARM` machines. We apply the ORDO primitive to various timestamp-based algorithms, which use either physical or logical timestamping, and scale these algorithms across various architectures by at most 39.7×, thereby making them multicore friendly.

References

[1] Exim Internet Mailer, 2015. http://www.exim.org/.

[2] ABALI, B., AND STUNKEL, C. B. Time Synchronization on SP1 and SP2 Parallel Systems. In *Proceedings of 9th International Parallel Processing Symposium* (Apr 1995), pp. 666–672.

[3] ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD/PODS Conference* (San Jose, CA, May 1995), pp. 23–34.

[4] AMD. Developer Guides, Manuals & ISA Documents, 2016. http://developer.amd.com/resources/developer-guides-manuals/.

[5] ARBEL, M., AND MORRISON, A. Predicate RCU: An RCU for Scalable Concurrent Updates. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)* (San Francisco, CA, Feb. 2015), pp. 21–30.

[6] ARM. The armv8-a architecture reference manual, 2016. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html.

[7] ATOOFIAN, E., AND BAVARSAD, A. G. AGC: Adaptive Global Clock in Software Transactional Memory. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores* (2012), PMAM '12, pp. 11–16.

[8] AVNI, H., AND SHAVIT, N. Maintaining Consistent Transactional States Without a Global Clock. In *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity* (2008), SIROCCO '08, pp. 131–140.

[9] B. WICKIZER, S., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. OpLog: a library for scaling update-heavy data structures. *CSAIL Technical Report* (2013).

[10] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Dec. 2008).

[11] BERNSTEIN, P. A., AND GOODMAN, N. Multiversion Concurrency Control&Mdash;Theory and Algorithms. *ACM Trans. Database Syst. 8*, 4 (Dec. 1983), 465–483.

[12] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[13] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Oct. 2010).

[14] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI.

[15] C. SEREBRIN, B. Fast, automatically scaled processor time stamp counter. https://patentscope.wipo.int/search/en/detail.jsf?docId=US42732523, 2009.

[16] C. SEREBRIN, B., AND M. KALLAL, R. Synchronization of processor time stamp counters to master counter. https://patentscope.wipo.int/search/en/detail.jsf?docId=US42732522, 2009.

[17] CARTA, A., LOCCI, N., MUSCAS, C., PINNA, F., AND SULIS, S. GPS and IEEE 1588 Synchronization for the Measurement of Synchrophasors in Electric Power Systems. *Comput. Stand. Interfaces 33*, 2 (Feb. 2011), 176–181.

[18] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency Without Ordering. In *10th USENIX Conference on File and Storage Technologies (FAST) (FAST 15)* (San Jose, CA, Feb. 2012).

[19] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA, June 2010), pp. 143–154.

[20] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012), pp. 251–264.

[21] CRISTIAN, F. Probabilistic clock synchronization. *Distributed Computing 3*, 3 (1989), 146–158.

[22] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA, Nov. 2013).

[23] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing (DISC)* (Stockholm, Sweden, Sept. 2006), pp. 194–208.

[24] DODDS, M., HAAS, A., AND KIRSCH, C. M. A Scalable, Correct Time-Stamped Stack. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)* (Mumbai, India, Jan. 2015), pp. 233–246.

[25] ELSON, J., GIROD, L., AND ESTRIN, D. Fine-grained Network Time Synchronization Using Reference Broadcasts. 147–163.

[26] FELBER, P., FETZER, C., AND RIEGEL, T. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and*

Practice of Parallel Programming (PPoPP) (Salt Lake City, UT, Feb. 2008), pp. 237–246.

[27] HAAS, A., KIRSCH, C., LIPPAUTZ, M., PAYER, H., PREISHUBER, M., ROCK, H., SOKOLOVA, A., HENZINGER, T. A., AND SEZGIN, A. Scal: High-performance multicore-scalable data structures and benchmarks, 2013. http://scal.cs.uni-salzburg.at/.

[28] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[29] HERMAN, N., INALA, J. P., HUANG, Y., TSAI, L., KOHLER, E., LISKOV, B., AND SHRIRA, L. Type-aware Transactions for Faster Concurrent Code. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)* (London, UK, Apr. 2016), pp. 31:1–31:16.

[30] INTEL. Intel 64 and IA-32 Architectures Software Developer Manuals, 2016. https://software.intel.com/en-us/articles/intel-sdm.

[31] INTEL. *Xeon Processor E7-8890 v4 (60M Cache, 2.20 GHz)*, 2016. http://ark.intel.com/products/93790/Intel-Xeon-Processor-E7-8890-v4-60M-Cache-2_20-GHz.

[32] JOHNSON, R., PANDIS, I., HARDAVELLAS, N., AILAMAKI, A., AND FALSAFI, B. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (2009), EDBT '09, pp. 24–35.

[33] JOHNSON, R., PANDIS, I., STOICA, R., ATHANASSOULIS, M., AND AILAMAKI, A. Aether: A Scalable Approach to Logging. 681–692.

[34] KIM, K., WANG, T., JOHNSON, R., AND PANDIS, I. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 ACM SIGMOD/PODS Conference* (San Francisco, CA, USA, 2016), pp. 1675–1687.

[35] KIMURA, H. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference* (Melbourne, Victoria, Australia, May 2015), pp. 691–706.

[36] KUNG, H. T., AND ROBINSON, J. T. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[37] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM 21*, 7 (July 1978), 558–565.

[38] LARSON, P.-A., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-performance Concurrency Control Mechanisms for Main-memory Databases. 298–309.

[39] LEE, C., SIM, D., HWANG, J.-Y., AND CHO, S. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST) (FAST 15)* (Santa Clara, CA, Feb. 2015), pp. 273–286.

[40] LEE, K. S., WANG, H., SHRIVASTAV, V., AND WEATHERSPOON, H. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 27th ACM SIGCOMM* (Florianopolis, Brazil, Aug. 2016), pp. 454–467.

[41] LEPERS, B., QUÉMA, V., AND FEDOROVA, A. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)* (Santa Clara, CA, July 2015), pp. 277–289.

[42] LEV, Y., LUCHANGCO, V., MARATHE, V. J., MOIR, M., NUSSBAUM, D., AND OLSZEWSKI, M. Anatomy of a scalable software transactional memory. In *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'09)* (2009).

[43] MARTIN G., D., J. SHRALL, J., PARTHASARATHY, S., AND RAJESH. Controlling time stamp counter (TSC) offsets for mulitple cores and threads. https://patentscope.wipo.int/search/en/detail.jsf?docId=US73280125o, 2011.

[44] MATVEEV, A., SHAVIT, N., FELBER, P., AND MARLIER, P. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)* (Monterey, CA, Oct. 2015), pp. 168–183.

[45] MCCRAKEN, D. Object-based Reverse Mapping.

[46] MCKENNEY, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf.

[47] MILLS, D. L. A Brief History of NTP Time: Memoirs of an Internet Timekeeper. *SIGCOMM Comput. Commun. Rev. 33*, 2 (Apr. 2003), 9–21.

[48] MIN, C., KASHYAP, S., MAASS, S., KANG, W., AND KIM, T. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)* (Denver, CO, June 2016).

[49] MINH, C. C. TL2-X86, 2015. https://github.com/ccaominh/tl2-x86.

[50] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst. 17*, 1 (Mar. 1992), 94–162.

[51] MOSER, H., AND SCHMID, U. Optimal Clock Synchronization Revisited: Upper and Lower Bounds in Real-Time Systems. In *Principles of Distributed Systems: 10th International Conference, OPODIS 2006, Bordeaux, France, December 12-15, 2006. Proceedings* (2006), Springer Berlin Heidelberg, pp. 94–109.

[52] NARULA, N., CUTLER, C., KOHLER, E., AND MORRIS, R. Phase Reconciliation for Contended In-memory Transactions. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014), pp. 511–524.

[53] ORACLE. *Data Sheet: SPARC M7-16 Server*, 2015. http://www.oracle.com/us/products/servers-storage/sparc-m7-16-ds-2687045.pdf.

[54] ORACLE. Oracle SPARC Architecture 2011, 2016. http://www.oracle.com/technetwork/server-

storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf.

[55] PAVLO, A., CURINO, C., AND ZDONIK, S. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD/PODS Conference* (Scottsdale, Arizona, USA, May 2012), pp. 61–72.

[56] RIEGEL, T., FELBER, P., AND FETZER, C. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Conference on Distributed Computing* (2006), DISC'06, pp. 284–298.

[57] RIEGEL, T., FETZER, C., AND FELBER, P. Time-based Transactional Memory with Scalable Time Bases. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (2007), SPAA '07, pp. 221–228.

[58] RUAN, W., LIU, Y., AND SPEAR, M. Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters. *ACM Transactions on Architecture and Code Optimization 10*, 4 (Dec. 2013), 40:1–40:21.

[59] SCHWEIZER, H., BESTA, M., AND HOEFLER, T. Evaluating the Cost of Atomic Operations on Modern Architectures. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (WA, DC, USA, Oct. 2015), pp. 445–456.

[60] SHAVIT, N., AND TOUITOU, D. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (1995), PODC '95, pp. 204–213.

[61] SRIKANTH, T. K., AND TOUEG, S. Optimal Clock Synchronization. *J. ACM 34*, 3 (July 1987), 626–645.

[62] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA, Nov. 2013), pp. 18–32.

[63] WANG, Q., STAMLER, T., AND PARMER, G. Parallel Sections: Scaling System-level Data-structures. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)* (London, UK, Apr. 2016), pp. 33:1–33:15.

[64] WANG, T., AND JOHNSON, R. Scalable Logging Through Emerging Non-volatile Memory. 865–876.

[65] YU, X. DBx1000, 2016. https://github.com/yxymit/DBx1000.

[66] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* (Nov. 2014), 209–220.

[67] YU, X., PAVLO, A., SANCHEZ, D., AND DEVADAS, S. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 ACM SIGMOD/PODS Conference* (San Francisco, CA, USA, 2016), pp. 1629–1642.

[68] YUAN, Y., WANG, K., LEE, R., DING, X., XING, J., BLANAS, S., AND ZHANG, X. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-memory Databases. *Proc. VLDB Endow. 9*, 6 (Jan. 2016), 504–515.

[69] ZHANG, R., BUDIMLIĆ, Z., AND SCHERER, III, W. N. Commit Phase in Timestamp-based Stm. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures* (2008), SPAA '08, pp. 326–335.