A Scalable Ordering Primitive for Multicore Machines

Sanidhya Kashyap Changwoo Min† Kangnyeon Kim* Taesoo Kim

Georgia Institute of Technology †*Virginia Tech* **University of Toronto*

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. 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 proposing and designing a scalable ordering primitive, called Ordo, that relies on *invariant* hardware clocks. Ordo not only enables the correct use of these clocks, by providing a notion of a global hardware clock, but also frees various logical timestamp-based algorithms from the burden of the software logical clock, while trying to simplify their design. 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 that there is a possibility that the clocks are not synchronized on two architectures (`Intel` and `ARM`) and that Ordo generally improves the efficiency of several algorithms by 1.2–39.7× on various architectures.

ACM Reference Format:

Sanidhya Kashyap Changwoo Min† Kangnyeon Kim* Taesoo Kim . 2018. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of Thirteenth European Conference on Systems (Eurosys'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3190508.3190510

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, consistency depends on algorithms that require linearizability [27] for the composability of a data structure [8, 44], concurrency control mechanisms for software transactional memory (`STM`) [21], or serializability (or snapshot isolation) for database transactions [36, 62, 67]. The notion of ordering not only is limited to consistency, but also is applicable to either maintaining the history of operations for logging [32, 39, 50, 64] or determining the quiescence period for memory reclamation [4, 44, 63]. Depending on the consistency requirement, ordering such as a logical timestamping [21, 44], physical timestamping [4, 8, 18, 22, 57], and data-driven versioning [37, 67] can be achieved in several ways. The most common approach is to use a logical clock that is easier to maintain by software and is amenable to various ordering requirements.

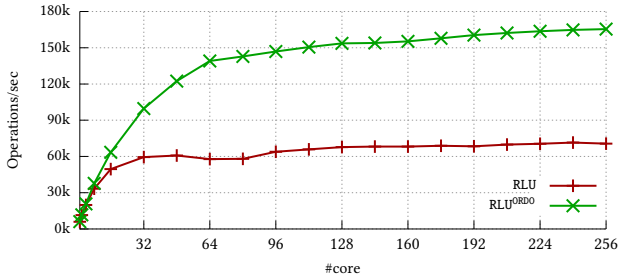

Figure 1: Throughput of read-log-update (`RLU`), a concurrency mechanism, on a hash table benchmark with 98% reads and 2% writes. While the original version (`RLU`) maintains a global logical clock, `RLU`$^{\texttt{ORDO}}$ is a redesign of `RLU` with our proposed Ordo primitive.

A logical clock, a favorable ordering approach, is one of the prime scalability bottlenecks on large multicore and multisocket machines [30, 52] 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 [12, 20, 58]. For example, a recently proposed synchronization mechanism for concurrent programming, called `RLU` [44], gets 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 an application, which holds true for several concurrency control mechanisms for concurrent programming [21, 23] 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 [9, 54]—have been put into practice. However, these approaches have some side effects. They either increase the abort rates for `STM` [7] and databases [62] or complicate the design of the system software [9, 67, 69]. In addition, in the past decade, physical timestamping has been gaining

Eurosys'18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190510

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

traction for designing scalable concurrent data structures [22, 25] and a synchronization mechanism [8] 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 [3, 5, 29, 53]. An invariant clock has a unique property: It is monotonically increasing and has a constant skew, regardless of dynamic frequency and voltage scaling, and resets itself to zero whenever a machine is reset (on receiving the RESET signal), which is ensured by the processor vendors [5, 15, 43, 53]. 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, which makes these clocks unusable. Thus, we cannot compare two clocks with confidence when correctly designing any time-based concurrent algorithms. Moreover, we cannot exactly synchronize these clocks because 1) the conventional clock algorithms [16, 40, 47] provide only a time bound that can have a lot of overhead, and 2) the hardware vendors do not divulge any communication cost that is required to establish strict time bounds for software clock synchronization to work effectively.

The comparison of clocks with no confidence makes them unusable to correctly design a concurrent algorithm. Thus, to provide a guarantee of correct use of these clocks, we propose a new primitive, called Ordo, that embraces uncertainty when we compare two clocks and provides an illusion of a *globally synchronized hardware clock* in a single machine. However, the notion of a globally synchronized hardware clock is only feasible if we can measure the offset between clocks. Unfortunately, accurately measuring this offset is difficult because hardware does not provide minimum bounds for the message delivery [16, 40, 47]. To solve this problem, we exploit the unique property of invariant clocks 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 be simplified as well as benefit from our Ordo primitive, which has led us to design a simple API. The only trick lies in handling the uncertainty window, which we explain for both physical and logical timestamp-based algorithms such as a 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) up to 39.7× across four architectures: `Intel Xeon` and `Xeon Phi`, `AMD`, and `ARM`, while maintaining equivalent performance in optimized scenarios. Moreover, our version of 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, that 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 simple API that almost all sets of algorithms can use.
- **Applications.** We apply Ordo to five concurrent algorithms and evaluate their performance on various architectures up to 256 hardware threads.

In the rest of the paper, we give a background and related prior works about timestamping (§2); then we present the correctness proof of the Ordo primitive (§3) and its applicability to various algorithms (§4). Later, we present our implementation (§5) and evaluate Ordo (§6). In the end, we discuss limitations and the future work of Ordo (§7), and then we conclude (§8).

2 BACKGROUND AND MOTIVATION

We first discuss the importance of timestamping with respect to prior research and briefly peruse invariant clocks provided by today's hardware.

2.1 Time: An Ingredient for Ordering

Timestamping, or versioning, is an essential tool to achieve ordering. Hence, on the basis of its applicability, we classify prior research directions in two categories: logical clocks and physical timestamping.

Logical clocks. Logical timestamping, or a software clock, is a prime primitive for most concurrency mechanisms in the domain of concurrent programming [4, 6, 7, 23, 28, 42, 44, 57] or database transactions [36, 38, 62, 66, 67]. To achieve ordering, these applications rely on atomic instructions, such as `fetch_and_add` [23, 28, 44, 60] or `compare_and_swap` [21]. For example, software transactional memory (`STM`) [59] is an active area of research in which almost every new design [28, 60] is dependent on time-based approaches [21, 55] that use global clocks for simplifying the validation step while committing a transaction. As mentioned before, a global clock incurs cache-line contention; thus, prior studies try to mitigate this contention by applying various forms of heuristics [6, 7] or slowly increasing timers [56], or having access to a synchronized hardware clock [57]. Unfortunately, these approaches introduce false aborts (heuristics) or are limited to specific hardware (synchronized clocks). The use of Ordo is inspired by prior studies that either used synchronized clocks [56] or relied on specific property to achieve causal ordering on a specific architecture such as Intel X86 [57]. However, Ordo assumes that it has access to only invariant timestamp counters with an uncertainty window, and we expose a set of methods for the user to handle that uncertainty regardless of the architecture. Similar to `STM`, `RLU` [44] is a lightweight synchronization mechanism that is an alternative to RCU [46]. It also employs a global clock that serializes writers and readers, and uses a defer-based approach to mitigate contention. We show that even with its defer-based approach, `RLU` suffers from the contention of the software clock, which we address with the Ordo primitive.

Databases rely on timestamping for concurrency control [10, 11, 36, 38, 62] to ensure a serializable execution schedule. Among concurrency control mechanisms, the optimistic approach is popular in practice [38, 62]. Yu *et al.* [66] evaluated the performance of

various major concurrency control mechanisms and emphasized that timestamp-based concurrency control mechanisms such as optimistic concurrency control (OCC) and multi-version concurrency control mechanisms (MVCC) suffer from timestamp allocation with increasing core count. Thus, various OCC-based systems have explored alternatives. For example, Silo [62] adopted a coarse-grain timestamping 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 does not scale in highly contentious workloads [34] due to its limited concurrency control. On the other hand, MVCC protocols still rely on logical clocks for timestamp allocation and suffer scalability collapse even in a read-only workload [66, 67]. Thus, Ordo overcomes the timestamping issue for both OCC and MVCC and is easily extendable to other timestamp-based CC schemes (refer to §6.5). Our work is inspired by the works of distributed transactions systems such as those of Thor [2], which used loosely synchronized clocks for concurrency control, and Spanner [18], which exposed the novel TrueTime API with uncertainty at the scale of data centers to handle externally consistent distributed transactions.

Physical timestamping. With the pervasiveness of multicore machines, physical timestamp-based algorithms are gaining more traction. For example, Oplog, an update-heavy data structure library, removes contention from a global data structure by maintaining a per-core log that appends operations in a temporal order [8]. Similarly, Dodds *et al.* [22] proposed a concurrent stack that scales efficiently with the RDTSC counter. Likewise, quiescence mechanisms have shown that clock-based reclamation [63] eschews the overhead of the epoch-based reclamation scheme. In addition, Recbulc [68] used core-local hardware timestamps to reduce the recording overhead of synchronization events and determine global ordering among them. They used a statistical approach to determine the skew between clocks, which is around 10 cycles for their machine. Unfortunately, besides Recbulc, the primary concern with these algorithms, is their assumption of access to synchronized clocks, which is not true with any available hardware. On the contrary, all clocks have constant skew and are monotonically increasing at a constant rate, which our Ordo primitive leverages to provide a globally synchronized clock on modern commodity hardware. Thus, with the help of our API, Ordo acts as a drop-in replacement for these algorithms. The only tweak that these algorithms require is to carefully handle the case of uncertainty that is present because of these invariant clocks. The idea proposed by Recbulc is not applicable in our scenario, as it still provides a statistical guarantee, which we want to avoid for designing concurrent algorithms that want to have the property of linearizability or serializability. Moreover, our approach ensures the correct use of these clocks with the help of one-way latency (§3).

2.2 A Primer on Hardware Clocks

While marking an operation, a physical clock provides an externally consistent view of operations in both distributed and multi-threaded environments. Unfortunately, it is difficult to establish an ordering between clocks because exactly synchronizing them is impossible

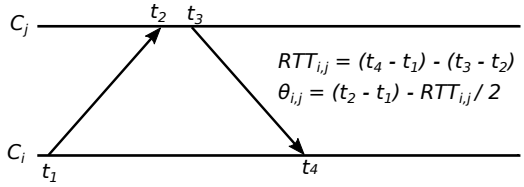

Figure 2: A simple clock-synchronization approach [47] to synchronize two clocks, c_i and c_j, that is used by various distributed systems. However, this approach is ineffective for synchronizing processor clocks because of the software overhead that leads to more coarse granularity of the clock synchronization protocol (in μs–ms) than the actual granularity of these clocks (in ns).

in practice [1, 18, 40]. Thus, to rely on these clocks, various distributed systems loosely synchronize them with the help of clock synchronization protocols [16, 40, 47, 61].

We observe a similar trend in multicore machines that provide per-core or per-processor hardware clocks [3, 5, 29, 53]. For example, all modern processor vendors such as Intel and AMD provide an RDTSC counter, while Sparc and ARM have stick and cntvct counter, respectively. These hardware clocks are **invariant** in nature, that is, processor vendors guarantee that these clocks are monotonically increasing at a constant rate,[2] regardless of the processor speed and its fluctuation, and they reset to zero whenever a processor receives a RESET signal (i.e., reboots). To provide such an invariant property, the current hardware always synchronizes these clocks from an external clock on the motherboard [14, 15, 43]. This is required because vendors guarantee that the clocks do not fluctuate even if a processor goes to deep sleep states, which is always ensured by the motherboard clock. However, vendors do no guarantee the synchronization of clocks across a processor (socket) boundary in a multicore machine [29], not even within a processor because there is no guarantee that each processor can receive the broadcasted RESET signal at the same instant either inside a socket or across them. We further confirm this trend in an Intel and ARM machine (refer to Figure 9 (a), (d)), in which one of the sockets has almost 4–8× higher measured offset than from the other direction. In addition, these hardware clocks do not provide the notion of real time.

These current invariant clocks are not reliable enough to design concurrent algorithms. To use them with confidence, we need to have a clock synchronization mechanism that provides constant physical skew between clocks, or we measure our offset to synchronize these clocks in software (refer to Figure 2). Unfortunately, none of these approaches work because hardware vendors cannot provide physical skew for every processor, and we cannot measure using existing clock synchronization protocols (refer to Figure 2), as software measurement induces overhead, in which the measured offset may be greater than the physical offset. Moreover, synchronizing clocks is challenging because we should be able to distinguish a few to tens of nano seconds difference in software (refer to Table 1). Thus, synchronizing clocks will result in mis-synchronized clocks, thereby leading to incorrect implementation of algorithms. To reliably use them, we devise an approach that

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

```
def get_time(): # Get timestamp without memory reordering
  return hardware_timestamp()  # Timestamp instruction

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: # t1 < t2
    return -1
  return 0                      # Uncertain

def new_time(time_t t): # New timestamp after ORDO_BOUNDARY
  while cmp_time(new_t = get_time(), t) is not 1:
    continue # pause for a while and retry
  return new_t # new_t is greater than (t + ORDO_BOUNDARY)
```

Figure 3: Ordo clock API. The `get_time()` method returns the current timestamp without reordering instructions.

embraces uncertainty by measuring a maximum possible offset that is still small enough in practice to ensure the correctness of these algorithms, which is contrary to existing clock synchronization algorithms [16, 40, 47, 61] or statistically calculating skew between them on a single machine [68].

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 an illusion, Ordo exposes a simple API that allows applications either to obtain a global timestamp or to order events with some uncertainty. Thus, we introduce an approach to measure the uncertainty window, followed by a proof to ensure its correctness.

3.1 Embracing Uncertainty in Clock: Ordo API

Timestamp-based concurrent algorithms can reliably use an invariant clock if we define the uncertainty period. Moreover, such algorithms are designed to execute on two or more cores/threads, which require two important properties from invariant clocks: 1) establishing a relation among two or more cores to compare events and 2) providing a notion of a monotonically increasing globally synchronized clock to order events in a single machine. Thus, we propose Ordo, which provides a notion of a monotonically increasing timestamp but also exposes an uncertainty window, called `ORDO_BOUNDARY`, in which we are unsure of the ordering. To ease the use of our Ordo primitive, we expose three simple methods (Figure 3) for such algorithms:

- **`get_time()`** is the hardware-specific timestamping instruction that also takes care of the hardware-specific requirements such as instruction reordering.
- **`new_time(time_t t)`** returns a new timestamp at the granularity of `ORDO_BOUNDARY` and is greater than `t`.
- **`cmp_time(time_t t1, time_t t2)`** establishes the precedence relation between timestamps with the help of `ORDO_BOUNDARY`. If the difference between `t1` and `t2` is within `ORDO_BOUNDARY`, it returns 0, meaning that we are uncertain.

```
runs = 100000  # multiple runs to minimize overheads
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() # synchronize 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,
                        max(clock_offset(c0, c1),
                            clock_offset(c1, c0)))
  return global_offset
```

Figure 4: Algorithm to calculate the `ORDO_BOUNDARY`: a system-wide global offset.

3.2 Measuring Uncertainty between Clocks: Calculating `ORDO_BOUNDARY`

Under the assumption of invariant clocks, the uncertainty window (or skew) between clocks is constant because both clocks monotonically increase at the same rate but may receive a 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 clocks, computes an offset, and then adjusts its clock by the measured offset [19]. However, this measurement introduces various errors, such as reading remote clocks and software overheads, including network jitter. Thus, we cannot rely on this method to define `ORDO_BOUNDARY` 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 [16, 40, 47];[3] and 3) periodically using the clock synchronization will waste CPU cycles, which will increase with core count. Moreover, the measured offset can be either larger or smaller than Δ, which renders it unusable for concurrent algorithms.

3.2.1 Measuring global offset. Instead of synchronizing these clocks among themselves, we exploit their unique *invariant* property and empirically calculate a system-wide global offset, called `ORDO_BOUNDARY`, which ensures that the measured offset between

[3] Although clock synchronization algorithms are widely used in distributed systems settings where multiple clocks are in use, they are also applicable in a single machine with per-core clocks, and it maintains the notion of a global clock by synchronizing itself with an outside clock over the network [47].

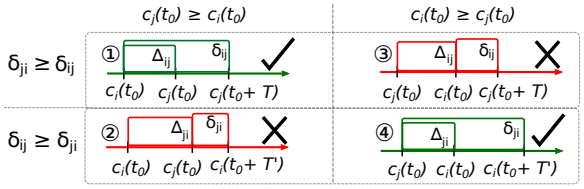

Figure 5: Calculating offset ($\delta_{i\leftrightarrow j}$) using pairwise one-way-delay latency between clocks (c_i and c_j). Δ_{ij} and Δ_{ji} are the physical offsets between c_i and c_j, and c_j and c_i, respectively. δ_{ij} and δ_{ji} are measured offsets with our approach. Depending on physical and measured offsets, there are four possible cases. Unlike existing clock-synchronization protocols [16, 19, 40, 47][3] that average the calculated latency based on RTT (refer to Figure 2), we consider each direction separately to measure the offset and consider only the direction that is always greater the positive physical offset, such as cases 1 and 4.

clocks is always greater than their physical offset. We define *measured offset* (δ_{ij}) as a one-way-delay latency between clocks (c_i to c_j), and this measured offset (δ) will be greater than the physical offset because of the extra one-way-delay latency.

Figure 4 illustrates our algorithm to calculate the global offset after calculating the correct pairwise offset for all clocks in a machine. We measure the offset between core c_i and c_j as follows: c_i atomically writes its timestamp value to the variable (line 8), which notifies waiting c_j (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, interrupts, and the coherence traffic. We reduce this error by calculating the offset multiple times (line 1) and taking the minimum of all runs (line 19–21). To define the correct offset between c_i and c_j, we calculate offset from both ends (i.e., δ_{ij} and δ_{ji}) and choose their maximum since we do not know which clock is ahead of the other (line 31). After calculating pairwise offsets among all cores, we select the maximum offset as the global offset, or `ORDO_BOUNDARY`, (line 30). We take the maximum offset among all cores because it ensures that any arbitrary core is guaranteed to see a new timestamp once the `ORDO_BOUNDARY` window is over, which enables us to compare 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 means of communication between cores.

3.2.2 Correctness of the measured offset. For invariant clocks to be useful, our approach to calculate the `ORDO_BOUNDARY` must have the following invariant:

> The global measured offset is always greater than the maximum physical offset between any cores.

We first state our assumptions and prove with a lemma and a theorem that our algorithm (Figure 4) to calculate the global offset is correct with respect to invariant clocks.

Assumptions. Our primary assumption is that clocks are invariant and have a physical offset that remains constant until the machine is reset. Moreover, the hardware's external clock-synchronization protocol maintains the monotonic increase of these clocks because it guarantees an invariant clock [14, 15, 43], which is a reasonable assumption for modern, mature many-core processors such as X86, Sparc, and ARM. Hence, with the invariant timestamp property, the physical offset, or skew, between clocks also remains constant.

Lemma. *The maximum of 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:

$$\delta_{ij} = |c_j(t_1) - c_i(t_0)| \qquad = |c_j(t_0 + T) - c_i(t_0)| \qquad (1)$$

$$\delta_{ji} = |c_i(t_1) - c_j(t_0)| \qquad = |c_i(t_0 + T') - c_j(t_0)| \qquad (2)$$

δ_{ij} and δ_{ji} denote the offset measured from c_i to c_j and c_j to c_i, respectively, and $|\Delta_{ij}| = |\Delta_{ji}|$. $c_i(t_0)$ 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 a message from another core (Figure 4: line 16), which denotes the one-way-delay latency in either direction (Figure 4: line 31). Depending on the relation between clocks at time t ($c_i(t)$ and $c_j(t)$) and measured offsets (δ_{ij} and δ_{ji}) from Equation 1 and 2, Figure 5 presents four possible cases: ① and ② if $c_j(t) \geq c_i(t)$, and ③ and ④ if $c_i(t) \geq c_j(t)$. Let us consider a scenario in which $c_j(t) \geq c_i(t)$ representing cases ① and ② and assume that we obtain case ②. By contradiction, case ② is not possible because our algorithm adds an extra cost of cache-line transfer (line 16) to measure the offset of constant monotonically increasing clocks, which will always return case ①; thus, δ_{ji} is going to be greater than δ_{ij}. Since we do not know which clock is lagging, we calculate other possible scenarios, ($c_i(t) \geq c_j(t)$: cases ③ and ④) so that we always obtain the maximum of two measured offsets without any physical offset information (lines 31–32). Thus, either case ① or case ④ always guarantees that measured offset is greater than the physical offset. □

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

Proof. $\delta^g = \max(\delta_{i\leftrightarrow j}) = \max(\max(\delta_{ij}, \delta_{ji})) \mid \forall i, j \in \{0..N\}$. N is the number of cores and δ^g is the `ORDO_BOUNDARY`. We extend the lemma to all pairwise combinations of cores (refer to Figure 4 line 29) to obtain the maximum offset value among all cores. This approach 1) establishes a relation among all cores such that if any core wants to have a *new timestamp*, it can obtain such a value at the expiration of `ORDO_BOUNDARY` and 2) also establishes an uncertainty period at a global level in which we can definitely provide the precedence relationship among two or more timestamps if all of them have a difference of the global offset; otherwise, we mark their comparison as uncertain if they fall in that offset window. □

4 ALGORITHMS WITH ORDO WITHOUT UNCERTAINTY

With the Ordo API, an important question is *how do we handle the uncertainty exposed by the Ordo primitive while adopting it for*

Logical Timestamping: CC Algorithm

```
txn_read/write(txn):
  txn.timestamp = get_timestamp()
  # Add to read/write set
txn_commit(txn):
  lock_writeset(txn)
  commit_txn = get_new_timstamp()
  if validate_readset(txn, commit_txn) is True:
      commit_write_changes(txn, commit_txn)
```

Physical Timestamping: oplog

```
op_log(op):
  local_log = get_local_log()
  thd.log.append([op, get_timestamp()])
op_synchronize(op):
  # Acquire per-object lock
  current_log = []
  for log in all_logs():
    # Add all op object to the current_log
  current_log.sort() # Sort via timestamp
  apply(current_log) # Apply all op objects
```

Figure 6: Pseudo-code of logical timestamping algorithms used in STM [21, 55] and databases [10, 36, 62], and that of physical timestamping used in Oplog [8].

timestamp-based algorithms? We answer this question by classifying them into two categories:

- *Physical to logical timestamping:* Algorithms that rely on logical timestamping or versioning (refer to Figure 6) will require all three methods, but the most important one is the `cmp_time()`, which provides the ordering between clocks to ensure their correctness. By introducing the `ORDO_BOUNDARY`, we now need to extend these algorithms to either defer or wait to execute any further operations.
- *Hardware timestamping:* Algorithms that use physical timestamping [8, 22] can use `new_time()` method to access a globally synchronized clock to ensure their correctness under invariant clocks.

4.1 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. It is inspired by STM [59] and maintains an object-level write-log per thread, in which writers first lock the object and then copy those objects to their own write log and manipulate them locally without affecting the original structure. RLU adopts the RCU barrier mechanism with a global clock-based logging mechanism [7, 21].

Figure 7 illustrates the pseudo-code of the functions of RLU that use the global clock. We refer to the pseudo-code to explain its workings, 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). For a 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 25), it increments the global clock (line 27), which effectively splits the memory snapshot into the old and new one. While old readers refer to the old snapshot that have smaller clock values than the incremented global clock, the new readers read the new snapshot that starts after the increment of the global clock (lines 18 – 22). Later, after increasing the global clock, writers wait for old readers to finish by executing the RCU-style quiescence loop (lines 41 – 50), while new operations obtain 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 31) and then releases the lock (line 32). In summary, RLU has three scalability bottlenecks: 1) maintain and reference

```
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 dereference
15   def rlu_dereference(ctx, obj):
16     ptr_copy = get_copy(obj) # Get pointer copy
17     # Return object or object copy
18     other_ctx = get_ctx(thread_id) # check for stealing
19 -   if other_ctx.write_clock <= ctx.local_clock:
20 +   if cmp_time(ctx.local_clock, other_ctx.write_clock) == 1:
21       return ptr_copy # return ptr_copy (stolen)
22     return obj # no stealing, return the oject
23 ------------------------------------------------------------
24   # Memory commit
25   def rlu_commit_write_log(ctx):
26 -   ctx.write_clock = global_clock + 1 # Enable stealing
27 -   fetch_and_add(global_clock, 1) # Advance clock
28 +   # Ordo clock with an extra ORDO_BOUNDARY for correctness
29 +   ctx.write_clock = new_time(ctx.local_clock + ORDO_BOUNDARY)
30     rlu_synchronize(ctx) # Drain readers
31     rlu_writeback_write_log(ctx) # Safe to write back
32     rlu_unlock_write_log(ctx)
33     ctx.write_clock = INFINITY # Disable stealing
34     rlu_swap_write_logs(ctx) # Quiesce write logs
35 ------------------------------------------------------------
36   # Synchronize with all threads
37   def rlu_synchronize(ctx):
38     for thread_id in active_threads:
39       other = get_ctx(thread_id)
40       ctx.sync_cnts[thread_id] = other.run_cnt
41     for thread_id in active_threads:
42       while:
43         if ctx.sync_cnts[thread_id] & 0x1 != 0:
44           break # not active
45         other = get_ctx(thread_id)
46         if ctx.sync_cnts[thread_id] != other.run_cnt:
47           break # already progressed
48 -       if ctx.writer_clock <= other.local_clock:
49 +       if cmp_time(other.local_clock, ctx.writer_clock) == 1:
50           break # started afer me
```

Figure 7: RLU pseudo-code including our changes.

the global clock, which does not scale with increasing core count, as shown in Figure 1; 2) lock/unlock operation on an object, and 3) copy an object for a 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 design choices that prefer programmability over the hand-crafted copy management mechanism.

We address the scalability bottleneck of the global clock with the Ordo primitive. Now, every read and write operation refers to the global clock via `get_time()` (line 6), and relies on it to dereference the object by comparing the timestamp for two contexts with the `cmp_time()` method (line 7), which provides the same comparison semantics before the modification (line 6). At the time of commit, we demarcate between the old and new snapshot by obtaining a new timestamp via the `new_time()` method (line 29), and later rely on this timestamp to maintain the clock-based quiescence period for readers that are still using the old snapshot (line 49). Note that

we add an extra ORDO_BOUNDARY to correctly differentiate between the old snapshot and the newer one, as we may obtain an incorrect snapshot if one of the clocks has negative skew.

Our modification does not break the correctness of the RLU algorithm, i.e., to always have a consistent memory snapshot in the RLU protected section. In other words, at time $t' > t$; because of the constant monotonically increasing clock, the time obtained at the commit time of the writer (line 29) is always greater than the previous write timestamp, thereby keeping a protected RLU section from seeing its concurrent overwrite. There can be an inconsistency if a thread has just updated a value and another thread is trying to steal the object while having a negative skew than the committed thread's clock. In this case, the reading thread may read an old snapshot, which can have an inconsistent memory snapshot. Since RLU only supports only a single version, we address this issue by adding an extra ORDO_BOUNDARY at the commit time, which ensures that we have at least one ORDO_BOUNDARY difference between or among threads. Moreover, our modification enforces the invariant for writers in the following ways: 1) If a new writer is able to steal the copy from the other writer (line 20), it still holds the invariant; 2) If a new writer is unable to steal the copy of the locked object (line 22), the old writer will quiesce while holding the writer lock (line 37), 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 versions of logs, which are swapped to allow stealing readers to use them (line 34).

4.2 Concurrency Control for Databases

Database systems serve highly concurrent transactions and queries, with strong consistency guarantees by using concurrency control (CC) schemes. Two main CC schemes are popular among state-of-the-art database systems: optimistic concurrency control (OCC) and multi-version concurrency control (MVCC). Both schemes use timestamps to decide global commit order without locking overhead [31]. Figure 6 presents the pseudo-code of CC algorithms.

OCC is a single CC scheme that consists of three phases: 1) *read*, 2) *validation*, and 3) *write*. In the first phase, a worker keeps footprints of a transaction in local memory. At commit time, it acquires locks on the write set. After that, it checks whether the transaction violates serializability in the *validation* phase by assigning a global commit timestamp to the transaction and validates both the read and write set by comparing their timestamps with the commit timestamp. After the *validation* phase, the worker enters the *write* phase in which it makes 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 with either conservative read validation [35, 62] or data-driven timestamping [67].

To show the impact of Ordo, we modify the first two phases—read and validation phases—of the OCC algorithm [36]. In the *read phase*, we assign the timestamp via `new_time()`, which guarantees that the new timestamp will be greater than the previous one. The validation scheme is the same as before, but the difference is that `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 both 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. Two requirements ensure serializability: 1) obtain a new timestamp that `new_time()` ensures, and 2) handle the uncertainty window, in which we conservatively abort transactions, thereby resolving the later-conflict check [2] to ensure the global ordering of transactions.

MVCC is another major category of CC schemes. Unlike other single-version CC schemes, MVCC takes an append-only update approach and maintains a version history, and uses timestamps to determine which version of records to serve. MVCC avoids reader-writer conflicts by forwarding them to physically different versions, thereby making this scheme more robust than OCC under high contention. To support time traveling and deciding the commit order, MVCC relies on logical timestamping, in the read and validation phase, which leads to severe scalability collapse (4.1–31.1× in Figure 13) due to the timestamp allocation with increasing core count [66]. We chose Hekaton as our example because it is the state-of-the-art in-memory database system [38]. Although it supports multiple CC levels with varying consistency guarantees, we focus on serializable, optimistic MVCC mode. We first describe the workings of the original algorithm and then introduce our modifications with the Ordo primitive.

Hekaton has the same three phases as OCC and works as follows: A worker in a transaction, reads the global clock at the beginning of the transaction. During the *read* stage, a transaction reads only a version when its begin timestamp falls between the begin and end timestamps[4] of the committed version. During update, a worker immediately appends its version to the database. The version consists of the transaction ID (TID) of the owner transaction marked in the begin timestamp field. At commit, the transaction assigns a commit timestamp to 1) determine the serialization order, 2) validate the read/scan sets, and 3) iterate the write set to replace TIDs installed in the begin timestamp field with that timestamp. Meanwhile, another transaction that encounters a TID-installed version examines visibility with the begin and end timestamps of the owner transaction.

We apply the Ordo primitive by first replacing the timestamp allocation to use `new_time` method in the beginning and during the commit phase of a transaction. 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. Thus, a worker proceeds only if the difference between timestamps is greater than the ORDO_BOUNDARY that provides a definite precedence relation. Otherwise, we either restart that transaction or force it to abort. However, given the small size of ORDO_BOUNDARY, we expect the aborts caused by this uncertainty to be rare. In terms of correctness, our approach provides the same consistency guarantee as the original one, i.e., 1) obtaining a unique timestamp, which `new_time()` ensures, and 2) maintaining the serial schedule, which is also maintained by `cmp_time()`, that only, conservatively, commits the transaction that has a precedence relation.

[4]Each version of records have begin timestamp, which indicates when the version becomes valid, and an end timestamp, which denotes when the version becomes invalid.

4.3 Software Transactional Memory (TL2)

We choose TL2 [21], 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 a global logical clock and checking the ownership record only once, i.e., it validates all 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 an orec[5] is unlocked and has no newer timestamp than the start timestamp of that transaction; it 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 of the locks in the write set and then validates all 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, denoting the end of the timestamp (*end*). It uses the *end* timestamp as a linearization 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 because a transaction atomically increments the global clock, which ensures the linearizability of a transactional update.

The basic requirement of the TL2 algorithm is that *end* should be greater than *start*, which the Ordo primitive (`new_time()`) ensures. Moreover, two disjoint transactions can share the same timestamp [70]. Thus, by using the Ordo API, we modify the algorithm as follows: We assign *start* with the value of `new_time()`, and use `new_time()` to obtain a definite newer timestamp for the *end* variable. Here, we again adopt a conservative approach of aborting transactions if two timestamps fall in the `ORDO_BOUNDARY`, as this can corrupt the memory or result in an undefined behavior of the program [21]. Although we can even use timestamp extension to remove aborts that occur during the read timestamp validation at the beginning of a transactional read, it may not benefit us because of the very small `ORDO_BOUNDARY`. Our modification ensures linearizability by 1) first providing an increasing timestamp via `new_time()` that globally provides a new timestamp value and 2) always validating the read set at the time of commit. In addition, we abort transactions if two timestamps (read set timestamp and commit timestamp) fall in the `ORDO_BOUNDARY` to remove uncertainty during the validation phase. Similarly, while performing a transactional load, we apply the same strategy to remove any inconsistencies.

4.4 Oplog: An Update-heavy Data Structures Library

Besides logical timestamping, physical timestamping is becoming common. In particular, there is an efficient implementation of a concurrent stack [22] and an update-heavy concurrency mechanism (Oplog [8]), with the same commonality of performing operations in a decentralized manner. We extend the Ordo primitive to Oplog. It maintains a per-core log, which stores update operations in a temporal order, and it applies logs on the centralized data structure in a sorted temporal order during the read phase. To ensure the correct temporal ordering across all per-core logs, Oplog requires a system-wide synchronized clock to determine the ordering of operations. We modify Oplog to use the `new_time()` method, which has a notion of a globally synchronized hardware clock while appending operations to a per-core log and use `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 across all invariant clocks. There is a possibility that two or more timestamps fall inside the `ORDO_BOUNDARY` during the merge phase, which denotes concurrent update operations, which is also possible in the original Oplog design. We address this problem by using the same technique of the original Oplog design, which is to apply these operations in an ascending order of the core id.

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

5 IMPLEMENTATION

Our library and micro benchmarks comprise 200 and 1,100 lines of code (LoC), in C, 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 the Ordo primitive and the architecture-specific code for ARM. To specifically evaluate database concurrency protocols, we choose DBx1000 [65] because it includes all database CC algorithms. We modify 400 LoC to support both OCC and Hekaton algorithms, including changes for the ARM architecture. We use an x86 port of the TL2 algorithm [49], which we extend (25 LoC) using the Ordo API.

6 EVALUATION

We evaluate the effectiveness of the Ordo primitive by answering the following key questions:

- How does an invariant hardware clock scale on various commodity architectures? (§6.1)
- What is the scalability characteristic of the Ordo primitive? (§6.2)
- What is the impact of the Ordo primitive on algorithms that rely on synchronized clocks? (§6.3)
- What is the 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 a higher number of physical cores and sockets, Phi has a higher degree of parallelism, and AMD is from a different processor vendor. These three processors have invariant hardware clocks in their specification. Moreover, we also use a 96-core ARM machine, whose clock is different from existing architectures. It supports a separate generic timer interface, which exists as a separate entity inside a processor [5]. We evaluate the scalability of clocks and algorithms up to the maximum number of hardware threads in a machine.

Machine	Cores	SMT	Speed (GHz)	Sockets	Offset between clocks	
					min (ns)	*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 configurations that we use in our evaluation as well as the calculated offset between cores. While *min* is the minimum offset between cores, *max* is the global offset, called ORDO_BOUNDARY (refer to Figure 4), which we used, including up to the maximum hardware threads (Cores∗SMT) in a machine.

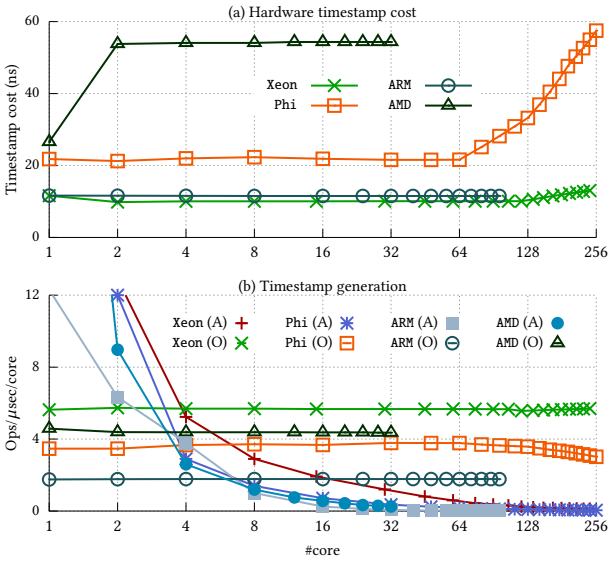

Figure 8: Micro evaluation of the invariant hardware clocks used by the ORDO primitive. (a) shows the cost of a single timestamping instruction when it is executed by varying the number of threads in parallel; (b) shows the number of per-core generated timestamps in a micro second with atomic increments (A) and with new_time() (O), which generates timestamps at each ORDO_BOUNDARY.

6.1 Scalability of Invariant Hardware Clocks

We create a simple benchmark to measure the cost of hardware timestamping instructions on various architectures. The benchmark forks a process on each core and repeatedly issues the timestamp instruction for a period of 10 seconds. Figure 8 shows the cost of the timestamp instruction on four different architectures. We observe that this cost remains constant up to the physical core count, but increases with increasing hardware threads, which is evident in Xeon and Phi. Still, it is 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 contention problem of the global clock with increasing core count, potentially together with hyperthreads.

6.2 Evaluating ORDO Primitive

Figure 9 presents the measured offset (δ_{ij}) for each pair-wise combination of the cores (with SMT). The heatmap shows that the measured offset between adjacent clocks inside a socket is the least on every architecture. One important point is that all 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 the past two months. This holds true with prior results [8, 57, 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 the 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 an application use in which the 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 may not be advantageous (§6.7). Therefore, we choose the global offset across all 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 maximum offset being 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 8 (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 transactions that use logical timestamping will definitely improve their scalability with increasing core count.

One key observation is that one of the sockets in both Xeon (eighth socket: 105–119 cores) and ARM (second socket: 48–96 cores) has a 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 architectures [41]. For example, the measured offset from core 50 to core 0 is 1,100 ns but is only 100 ns from core 0 to core 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. For Phi, we observe that most of the offset lies in the window of 200 ns, but adjacent cores have the least offset.

6.3 Physical Timestamping: Oplog

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

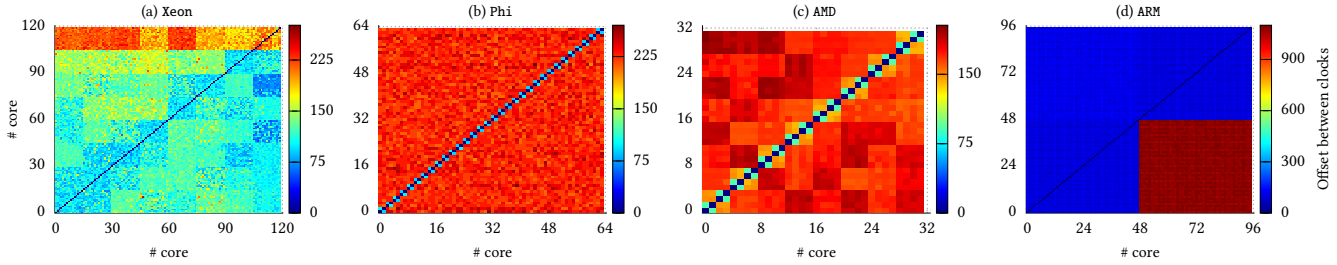

Figure 9: Clock offsets for all pairs of cores. The measured offset varies from a minimum of 70 ns to 1,100 ns for all architectures. Both Xeon (a) and ARM (d) machines show that the one of the sockets has a 4–8× higher offset than the others. To confirm this, we measured the bandwidth between the sockets, which is symmetric in both machines.

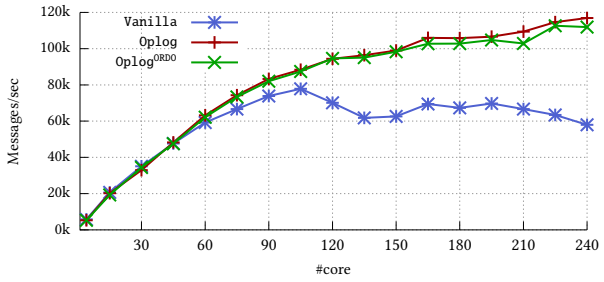

Figure 10: Throughput of Exim mail-server on a 240-core machine. Vanilla represents the unmodified Linux kernel, while Oplog is the rmap data structure modified by the Oplog API in the Linux kernel. Oplog^{ORDO} is the extension of Oplog with the Ordo primitive.

rmaps in the Linux kernel [8]. We use Exim mail-server [26] on the Xeon machine to evaluate the scalability of the rmap data structure. Exim is a part of the Mosbench [13] benchmark suite; it is a process-intensive application that 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 10 shows the results of Exim on the stock Linux (Vanilla) and our modifications that include kernel versions with (Oplog^{ORDO}) and without the Ordo primitive (Oplog). The Oplog version directly reads the value from the unsynchronized hardware clocks. 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^{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. We do not observe any huge difference between Oplog and Oplog^{ORDO} because a reverse mapping is updated only when a system call is issued, which amortizes the cost ORDO_BOUNDARY window, besides other virtual file system layer overhead [48].

6.4 Read Log Update

We evaluate the throughput of RLU for the hash table benchmark and citrus tree benchmark across architectures. We report only the numbers for the hash table benchmark, as we observe the same improvement with RLU^{ORDO} (almost 2×) for the citrus tree benchmark, involving complex update operations, across the 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 11 shows the throughput of the hash table with varying update rates of 2% and 40% across the architectures, where RLU is the original implementation and RLU^{ORDO} is the modified version. The results show that RLU^{ORDO} outperforms the original version by an average of 2.1× across 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 for RLU^{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^{ORDO}, and 2) RLU^{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^{ORDO} is 8% slower than RLU at the highest core across the architectures, but even with a 2% update rate, it is the atomic update that degrades the performance of RLU. On further analysis, we find that the RLU^{ORDO} spends at most 20% less time in the synchronize phase, which happens because of the decrease in the cache-coherence traffic on higher core count across all architectures. Furthermore, our Ordo's new_time() does not act as a backoff mechanism. Figure 16 illustrates that even on varying the ORDO_BOUNDARY boundary by 1/8×–8×, the scalability of RLU algorithm changes by only ±3% while running on 1-core, 1-socket, and 8-sockets.

Overall, all multisocket machines show a similar scalability trend after crossing the socket boundary and are saturated after a 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 is evident after 48 cores. In the case of Phi, there is no scalability collapse because it has a higher memory bandwidth and 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× in each case. Even though the cost of the 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 to Figure 12), RLU^{ORDO} is at most 1.8× faster than the

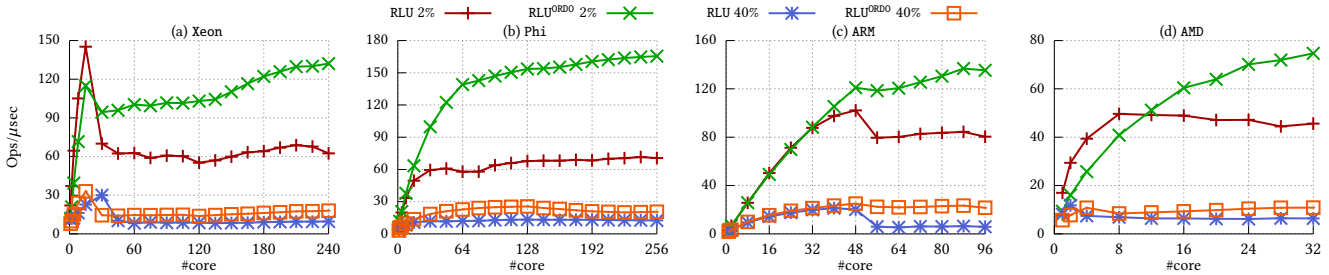

Figure 11: Throughput of the hash table with RLU and RLU^{ORDO} for various update ratios of 2% and 40% updates. The user space hash table has 1,000 buckets with 100 nodes. We experiment it on four machines from Table 1.

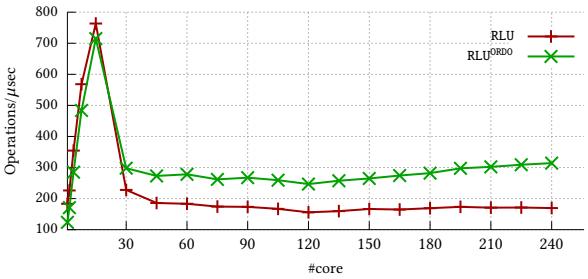

Figure 12: Throughput of the hash table benchmark (40% updates) with the deferred-based RLU and RLU^{ORDO} approach on the Xeon machine. Even with the deferred-based approach, the cost of the global clock is still visible after crossing the NUMA boundary.

base version, thereby illustrating that the defer-based approach still suffers from the contention on the global clock.

6.5 Concurrency Control Mechanism

We evaluate the impact of Ordo primitive on the existing OCC and Hekaton algorithms with YCSB [17] and TPC-C benchmarks. We execute read-only transactions to focus only on the scalability aspect without transaction contentions (Figure 13), which comprises two read-queries per transaction and a uniform random distribution. We also present results from the TPC-C benchmark with 60 warehouses as a contentious case (Figure 14). Figure 13 shows that both OCC^{ORDO} and Hekaton^{ORDO} not only outperform OCC (5.6–39.7×) and Hekaton (4.1–31.1×), respectively across architectures, but also achieve almost similar scalability as that of TicToc and Silo, which do not have global timestamp allocation overhead. The reason for such an improvement is that both OCC and Hekaton waste 62–80% of their execution time in allocating the timestamps, which has also been shown by Yu *et al.* [66], 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 these algorithms. In addition, Hekaton^{ORDO} has comparable performance to that of other OCC-based algorithms and is only 1.2–1.3× slower because of its heavyweight dependency-tracking mechanism that maintains multiple versions.

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 a 9% lower abort rate, since TicToc starts to spend more time (7%) in the validation phase because of the overhead of its data-driven timestamp computation, as it has to traverse the read and write set to find the common commit timestamp; OCC^{ORDO} already has a notion of global time, which speeds up the validation process, thereby increasing its throughput with lower aborts. Thus, hardware clocks provide better scalability than software bypasses that come at the cost of extra computation. In the case of multi-version CC, Hekaton^{ORDO} also outperforms Hekaton by 1.95× with lower 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 15 presents the speedup over the sequential execution of these benchmarks. The results show that TL2^{ORDO} improves the throughput of every benchmark. 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 stress the global clock with an increasing core count. In the case of Intruder, we observe some performance improvement up to 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 [28]. In the case of Labyrinth, we observe that the TL2^{ORDO} improves the throughput by 2–3.8×. Although Labyrinth has long running transactions [49], we observe that the number of aborts decreases by 5.8–15.5×, which illustrates that transactions in Labyrinth spend almost twice the amount of time in re-executing the aborted transactions, which occurs because of the cache-line contention of the global clock. 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 on 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.

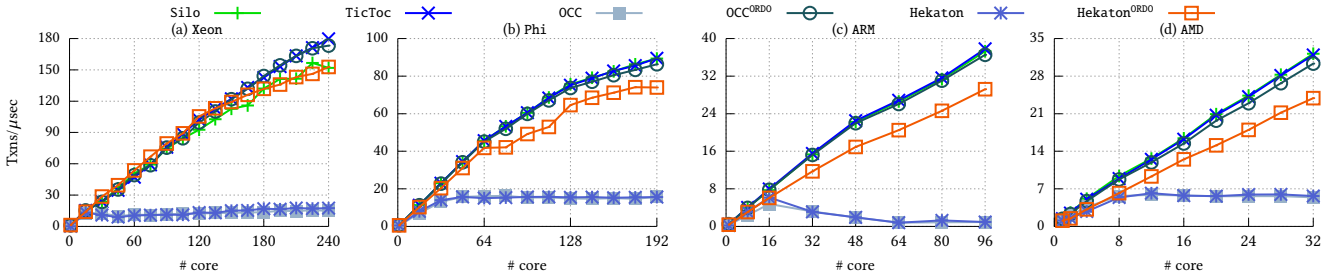

Figure 13: Throughput of various concurrency control algorithms of databases for read-only transactions (100% reads) using the YCSB benchmark on various architectures. We modify the existing OCC and Hekaton algorithms to use our ORDO primitive (OCC^{ORDO} and Hekaton^{ORDO}, respectively) and compare them against the state-of-the-art OCC algorithms: Silo and TicToc. Our modifications removes the logical clock bottleneck across various architectures.

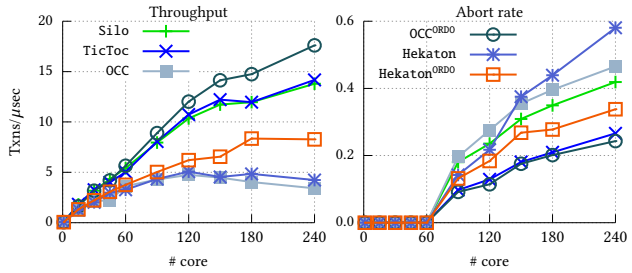

Figure 14: Throughput and abort rates of concurrency control algorithms for TPC-C benchmark with 60 warehouses on 240 Intel Xeon machine.

6.7 Sensitivity Analysis of ORDO_BOUNDARY

To show that the calculated ORDO_BOUNDARY does not hamper the throughput of the algorithm, we ran the hash table benchmark with the RLU^{ORDO} algorithm on the Xeon machine with 98% reads and 2% writes for three configurations: 1-core, 1-socket (30 cores), and 8-sockets (240 cores). We found that on either increasing or decreasing the ORDO_BOUNDARY by 1/8× or up to 8× of the global offset (refer to Table 1), the scalability of RLU^{ORDO} algorithm only varies by ±3% in all three configurations because the algorithm has other bottlenecks that overshadow the cost of both the lower and higher offsets. We can further confirm this result from Figure 8 (b) results, as hardware clocks can generate 1.4 billion timestamps at 240 cores, while the maximum throughput of the hash table is only 102 million for a 2% update operation. We observe a similar trend on a single core as well as for multiples of sockets. Furthermore, we can also infer that another important point is that the ORDO's new_time() method does not act as a backoff mechanism, as we can see consistent throughput even on decreasing the ORDO_BOUNDARY to 1/8×, which is merely 34 ns on the Xeon machine. This point also holds true for other timestamp algorithms, as they have to perform other tasks [49, 66].

7 DISCUSSION

In this section, we discuss the limitations and implications of ORDO and the invariant clocks with respect to the hardware, basic problems with clocks, and the design of algorithms.

Limitations. The most important issue with these hardware clocks is the instruction ordering that varies with architectures. For example, in the case of Intel architectures, we avoided instruction reordering by relying on the RDTSCP instruction followed by an atomic instruction to assign the timestamp. We do a similar operation for the ARM architecture as well. Another important problem with the timestamp counters is that they will overflow after crossing the 64-bit boundary in the case of Intel, AMD, and Sparc and the 55-bit boundary for ARM. Even though it will take years for the counters to overflow, the algorithms should also handle this scenario like existing approaches [23].

Another important issue with ORDO is that we assume that existing hardware clocks have constant skew and do not suffer from clock drift. However, this issue has been known to occur in various older machines [24]. On the contrary, we have not seen such an issue on any of our machines. Moreover, in a private communication, an Intel developer stated that current generations of Intel processors do not suffer from clock drift as the hardware tries to synchronize it. On the contrary, Oplog [8] empirically found that the clocks are synchronized. Since hardware vendors do not provide any such guarantee, our calculation is of the ORDO_BOUNDARY relaxes the notion of synchronized clocks while only assumes that they have constant skew. In addition, we believe that our algorithm (Figure 4) is applicable to machines with asymmetric architectures, such as AMD Bulldozer [33]. However, we could not evaluate the scalability of algorithms and the range of the ORDO_BOUNDARY because of the unavailability of the machine, but we believe that algorithms should still be scalable enough, as they have other bottlenecks besides timestamping.

Design of timestamp-based algorithms. Timestamping is one of the requirements of any logical timestamping-based algorithm. ORDO drastically improves the scalability of several algorithms across different architectures. On the other hand, ORDO's notion of global time simplifies these algorithms. For instance, a lot of prior works tried to reduce the cache-line contention in both databases [62] and STM [6, 7, 42, 44]. From the perspective of algorithm design, we

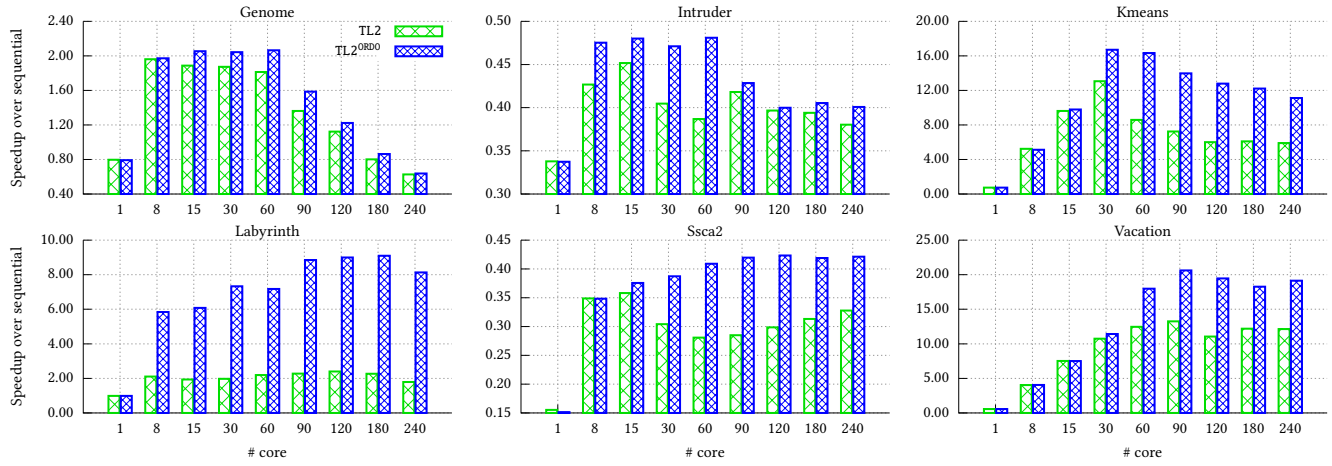

Figure 15: Speedup of the STAMP benchmark with respect to the sequential execution on `Xeon` machine for `TL2` and `TL2`$^{\texttt{ORDO}}$ algorithms. `TL2`$^{\texttt{ORDO}}$ improves the throughput up to 3.8× by alleviating the cache-line contention that occurs of the global logical clock. `TL2`$^{\texttt{ORDO}}$ shows significant improvement in the case of workloads running with very short transactions (Kmeans and Ssca2) and the ones with very long transactions by decreasing their aborts due to the cache-line mitigation (Labyrinth).

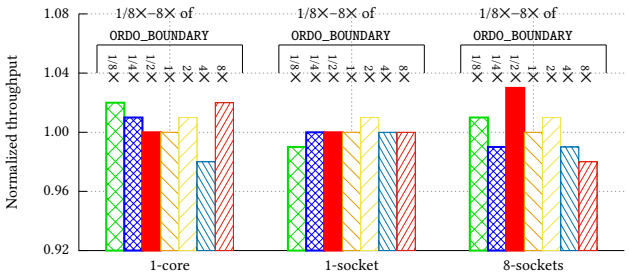

Figure 16: Normalized throughput of the `RLU`$^{\texttt{ORDO}}$ algorithm for varying `ORDO_BOUNDARY` on the `Xeon` machine on 1-core, 1-socket (30 cores), and 8-sockets (240 cores) with 98% reads and 2% writes. We vary the `ORDO_BOUNDARY` from 1/8×–8× for all three configurations, which shows that the throughput varies by only ±3%, thereby proving two points: 1) timestamping is one of the bottlenecks, and 2) `ORDO_BOUNDARY` does not act as a backoff mechanism for such logical timestamping-based algorithms.

can incorporate a new method to compare time based on the thread ID that can further decrease the uncertainty period. However, we avoided such a method for three reasons. First, maintaining a pairwise clock table will incur memory overhead and the application has to load the whole table in the memory for reading. Second, application developers may have to resort to thread pinning so that threads do not move while comparing or doing an operation, as thread migration can break the assumptions of algorithms. Finally, the value of `ORDO_BOUNDARY` is large enough to work on multisocket machines. Moreover, `ORDO_BOUNDARY` overcomes the problem of thread migration because the cost of thread migration varies from 1–200 μs, which does not affect the correctness of existing algorithms, as the time obtained after thread migration is already greater than the `ORDO_BOUNDARY` (refer to Table 1). However, Ordo is not a panacea to solving the timestamping issue. For instance, the timestamped stack [22] is oblivious to weak forms of clock synchronization that have stutter.

More hardware support. 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 a thousand cores. It is possible to further reduce the uncertainty window if processor vendors can provide some bound on the cost of cache-line access, which will allow us to use existing clock synchronization protocols with confidence, thereby decreasing the uncertainty window to overcome aborts occurring in concurrency algorithms. Moreover, other hardware vendors should provide invariant clocks that will enable applications to easily handle the ordering in a multicore-friendly manner.

Opportunities. The notion of a globally synchronized hardware clock opens a plethora of opportunities in various areas. For example, applications like Doppel [51] and write-ahead-logging [39] schemes can definitely benefit from the Ordo primitive. Since, an OS can easily measure the offset of invariant clocks, it is not applicable to virtual machines (VM). We believe that the hypervisor should expose this information to VMs to take full advantage of invariant hardware clocks, which cloud providers can easily expose through a paravirtual interface. 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 concurrency on large multicore and multisocket 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 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 several 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.

9 ACKNOWLEDGMENT

We thank the anonymous reviewers of both Eurosys 2018 and SOSP 2017, and our shepherd, José Pereira, for their helpful feedback. This research was supported, in part, by the NSF award DGE-1500084, CNS-1563848, CNS-1704701 and CRI-1629851, ONR under grant N000141512162, DARPA TC (No. DARPA FA8650-15-C-7556), Institute of Information & Communications Technology Promotion grant funded by the Korea government under contract ETRI MSIP/IITPB[2014-0-00035], and gifts from Facebook, Mozilla, and Intel.

REFERENCES

[1] B. Abali and C. B. Stunkel. 1995. Time Synchronization on SP1 and SP2 Parallel Systems. In *Proceedings of 9th International Parallel Processing Symposium*. IEEE, Piscataway, New Jersey, US, 666–672.

[2] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD/PODS Conference*. ACM, San Jose, CA, 23–34.

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

[4] Maya Arbel and Adam Morrison. 2015. Predicate RCU: An RCU for Scalable Concurrent Updates. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, San Francisco, CA, 21–30.

[5] ARM. 2016. The ARMv8-A Architecture Reference Manual. (2016). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html.

[6] Ehsan Atoofian and Amir Ghanbari Bavarsad. 2012. AGC: Adaptive Global Clock in Software Transactional Memory. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '12)*. ACM, New York, NY, USA, 11–16.

[7] Hillel Avni and Nir Shavit. 2008. Maintaining Consistent Transactional States Without a Global Clock. In *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity (SIROCCO '08)*. Springer-Verlag, Berlin, Heidelberg, 131–140.

[8] Silas B. Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2013. OpLog: a library for scaling update-heavy data structures. *CSAIL Technical Report* 1, 1 (2013), 1–12.

[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2008. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, San Diego, CA, 29–44.

[10] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control&Mdash;Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983), 465–483.

[11] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Vancouver, Canada, 1–16.

[13] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Vancouver, Canada, 1–16.

[14] Benjamin C. Serebrin. 2009. Fast, automatically scaled processor time stamp counter. https://patentscope.wipo.int/search/en/detail.jsf?docId=US42732523. (2009).

[15] Benjamin C. Serebrin and Robert M. Kallal. 2009. Synchronization of processor time stamp counters to master counter. https://patentscope.wipo.int/search/en/detail.jsf?docId=US42732522. (2009).

[16] Andrea Carta, Nicola Locci, Carlo Muscas, Fabio Pinna, and Sara Sulis. 2011. GPS and IEEE 1588 Synchronization for the Measurement of Synchrophasors in Electric Power Systems. *Comput. Stand. Interfaces* 33, 2 (Feb. 2011), 176–181.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, Indianapolis, Indiana, USA, 143–154.

[18] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Hollywood, CA, 251–264.

[19] Flaviu Cristian. 1989. Probabilistic clock synchronization. *Distributed Computing* 3, 3 (1989), 146–158.

[20] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. 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)*. ACM, Farmington, PA, 33–48.

[21] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing (DISC)*. Springer-Verlag, Berlin, Heidelberg, 194–208.

[22] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*. ACM, Mumbai, India, 233–246.

[23] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, Salt Lake City, UT, 237–246.

[24] Thomas Gleixner. 2010. RE: [PATCH] x86: Export tsc related information in sysfs . (2010). https://lwn.net/Articles/388286/.

[25] Andreas Haas, Christoph Kirsch, Michael Lippautz, Hannes Payer, Mario Preishuber, Harald Rock, Ana Sokolova, Thomas A. Henzinger, and Ali Sezgin. 2013. Scal: High-performance multicore-scalable data structures and benchmarks. (2013). http://scal.cs.uni-salzburg.at/.

[26] Philip Hazel. 2015. Exim Internet Mailer. (2015). http://www.exim.org/.

[27] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.

[28] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware Transactions for Faster Concurrent Code. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. ACM, London, UK, 31:1–31:16.

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

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

[31] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. 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 (EDBT '09)*. ACM, New York, NY, USA, 24–35.

[32] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, Singapore, 681–692.

[33] David Kantner. 2011. AMD's Bulldozer Microarchitecture. (2011). https://www.realworldtech.com/bulldozer/9/.

[34] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 ACM SIGMOD/PODS Conference*. ACM, San Francisco, CA, USA, 1675–1687.

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

[36] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226.

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

[38] PerAke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2012. High-performance Concurrency Control Mechanisms for Main-memory Databases. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, Istanbul, Turkey, 298–309.

[39] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST) (FAST 15)*. USENIX Association, Santa Clara, CA, 273–286.

[40] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 27th ACM SIGCOMM*. ACM, Florianopolis, Brazil, 454–467.

[41] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*. USENIX Association, Santa Clara, CA, 277–289.

[42] Yossi Lev, Victor Luchangco, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. 2009. Anatomy of a scalable software transactional memory. In *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'09)*. ACM, New York, NY, USA, 1–10.

[43] Dixon Martin G., Jeremy J. Shrall, S. Parthasarathy, and Rajesh. 2011. Controlling time stamp counter (TSC) offsets for mulitple cores and threads. https://patentscope.wipo.int/search/en/detail.jsf?docId=US73280125o. (2011).

[44] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Monterey, CA, 168–183.

[45] Dave McCraken. 2004. Object-based Reverse Mapping. *Proceedings of the Linux Symposium* 2, 1 (2004), 1–6.

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

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

[48] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. USENIX Association, Denver, CO, 71–85.

[49] Chi Cao Minh. 2015. TL2-X86. (2015). https://github.com/ccaominh/tl2-x86.

[50] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162.

[51] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended In-memory Transactions. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Broomfield, Colorado, 511–524.

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

[53] Oracle. 2016. Oracle SPARC Architecture 2011. (2016). http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf.

[54] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD/PODS Conference*. ACM, Scottsdale, Arizona, USA, 61–72.

[55] Torvald Riegel, Pascal Felber, and Christof Fetzer. 2006. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Conference on Distributed Computing (DISC'06)*. Springer-Verlag, Berlin, Heidelberg, 284–298.

[56] Torvald Riegel, Christof Fetzer, and Pascal Felber. 2007. Time-based Transactional Memory with Scalable Time Bases. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07)*. ACM, New York, NY, USA, 221–228.

[57] Wenjia Ruan, Yujie Liu, and Michael Spear. 2013. Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters. *ACM Transactions on Architecture and Code Optimization* 10, 4, Article 40 (Dec. 2013), 21 pages.

[58] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the Cost of Atomic Operations on Modern Architectures. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, WA, DC, USA, 445–456.

[59] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 204–213.

[60] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 682–696.

[61] T. K. Srikanth and Sam Toueg. 1987. Optimal Clock Synchronization. *J. ACM* 34, 3 (July 1987), 626–645.

[62] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Farmington, PA, 18–32.

[63] Qi Wang, Timothy Stamler, and Gabriel Parmer. 2016. Parallel Sections: Scaling System-level Data-structures. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. ACM, London, UK, 33:1–33:15.

[64] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging Through Emerging Non-volatile Memory. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, Hanghzou, China, 865–876.

[65] Xiangyao Yu. 2016. DBx1000. (2016). https://github.com/yxymit/DBx1000.

[66] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, Hanghzou, China, 209–220.

[67] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 ACM SIGMOD/PODS Conference*. ACM, San Francisco, CA, USA, 1629–1642.

[68] Xiang Yuan, Chenggang Wu, Zhenjiang Wang, Jianjun Li, Pen-Chung Yew, Jeff Huang, Xiaobing Feng, Yanyan Lan, Yunji Chen, and Yong Guan. 2015. ReCBuLC: Reproducing Concurrency Bugs Using Local Clocks. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 824–834. http://dl.acm.org/citation.cfm?id=2818754.2818854

[69] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-memory Databases. *Proc. VLDB Endow.* 9, 6 (Jan. 2016), 504–515. https://doi.org/10.14778/2904121.2904126

[70] Rui Zhang, Zoran Budimlić, and William N. Scherer, III. 2008. Commit Phase in Timestamp-based Stm. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '08)*. ACM, New York, NY, USA, 326–335.