MIT Libraries | DSpace@MIT

MIT Open Access Articles

Performant almost-latch-free data structures using epoch protection in more depth

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Li, T., Chandramouli, B. & Madden, S. Performant almost-latch-free data structures using epoch protection in more depth. The VLDB Journal (2024).

As Published: 10.1007/s00778-024-00859-8

Publisher: Springer Science and Business Media LLC

Persistent URL: https://hdl.handle.net/1721.1/155305

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of use: Creative Commons Attribution

Massachusetts Institute of Technology

The VLDB Journal
https://doi.org/10.1007/s00778-024-00859-8

SPECIAL ISSUE PAPER

Performant almost-latch-free data structures using epoch protection in more depth

Tianyu Li[1] · Badrish Chandramouli[2] · Samuel Madden[1]

Received: 15 March 2023 / Revised: 28 December 2023 / Accepted: 10 May 2024
© The Author(s) 2024

Abstract
Multi-core scalability presents a major implementation challenge for data system designers today. Traditional methods such as latching no longer scale in today's highly parallel architectures. While the designer can make use of techniques such as latch-free programming to painstakingly design specialized, highly-performant solutions, such solutions are often intricate to build and difficult to reason about. Of particular interest to data system designers is a class of data structures we call *almost-latch-free*; such data structures can be made scalable in the common case, but have rare complications (e.g., dynamic resizing) that prevent full latch-free implementations. In this work, we present a new programming framework called Epoch-Protected Version Scheme (EPVS) to make it easy to build such data structures. EPVS makes use of *epoch protection* to preserve performance in the common case of latch-free operations, while allowing users to specify critical sections that execute under mutual exclusion for the rare, non-latch-free operations. We showcase the use of EPVS-based concurrency primitives in a few practical systems to demonstrate its competitive performance and intuitive guarantees. EPVS is available in open source as part of Microsoft's FASTER project (Epoch Protected Version Scheme (source code) 2022; Microsoft FASTER 2022).

Keywords Multi-core programming · Concurrency control · Epoch protection

1 Introduction

Modern processors have seen a steady increase in core counts over the past several decades. Consequently, modern applications use many more threads, which makes safe and scalable concurrent access to shared data structures crucial for application correctness and performance. Traditional solutions such as mutexes or reader–writer latches scale poorly under this environment, scaling up to only a handful of threads [16, 23]. To alleviate these bottlenecks, modern systems use a wide range of techniques for latch-free programming. However, building latch-free systems and reasoning about their correctness is notoriously difficult; in our own experience building FASTER [8], Silo [49], and NoisePage [30], doing so requires months of careful design and many lines of subtle code, followed by lengthy debugging sessions. The main reason behind this difficulty is that latch-free systems rely on atomic primitives provided by the hardware (e.g., compare-and-swap instructions) instead of latches. Consequently, unlike with latch-based programming, developers cannot easily define critical sections for exclusion and must reason about numerous interleavings on the instruction level to establish invariants and ensure correctness. In short, developers today face a steep trade-off between intuitive latch-based programming that does not scale, and scalable latch-based programming that is difficult to build and maintain.

Such a trade-off is particularly costly for a common class of use cases we call "almost-latch-free". Almost-latch-free data structures can be accessed and modified latch-free in most cases, but must rely on more complex mechanisms in rare situations for correct synchronization. Examples of this include latch-free hash tables or arrays that occasionally resize, and metadata data structures that are mostly read-only but are occasionally updated. Ideally, developers implement latch-free programming for the common case and revert to

✉ Tianyu Li
litianyu@csail.mit.edu

Badrish Chandramouli
badrishc@microsoft.com

Samuel Madden
madden@csail.mit.edu

[1] MIT CSAIL, Cambridge, MA, USA

[2] Microsoft Research, Redmond, WA, USA

Published online: 17 June 2024

latch-based solutions in rare cases, but they are unable to easily do so with today's primitives.

In this paper, we present a general solution for building almost-latch-free data structures, called Epoch Protected Version Scheme (EPVS). EPVS offers lightweight, epoch [27] protection that allows programmers to mix common latch-free operations with strong, mutually exclusive critical sections that execute without any interleaving latch-free operations. Similar to latches, users protect code blocks and specify the protection level (latch-free or mutually-exclusive) for each block, and EPVS ensures that each code block is executed at the specified protection level. At the heart of EPVS is a highly optimized epoch protection framework from the FASTER system [8] that can sustain tens of millions of fine-grained operations per second on a modern multi-core machine. The simplest way to use EPVS is as a replacement for the standard reader–writer latch with much more scalable shared access but more expensive exclusive access. For advanced users, EPVS allows complex orchestration of concurrent operations and long-running critical sections through a general state machine model. For example, [43] and [30] both implemented bespoke almost-latch-free solutions that can be simplified with EPVS. Our experience shows that EPVS can deliver competitive performance and is intuitive to use.

Summary of contributions

- We identify almost-latch-free data structures as a common challenge in building scalable data systems.
- We present EPVS, a general solution to the almost-latch-free problem, which makes use of epoch protection to achieve highly scalable operation and strong mutual exclusion properties where necessary.
- We evaluate EPVS in the context of practical data system challenges to demonstrate its performance and ease of use.

2 Background

2.1 Almost-latch-free data structures

As mentioned, almost-latch-free data structures are those that are amenable to simple, efficient, and scalable latch-free implementations, but under rare conditions must resort to more complex thread coordination, which in turn prevents latch-free operations in the common case as well. For example, consider a simple resizable array (holding 8-byte object references) implementation, with only the following methods for simplicity of illustration:

- `Count()`: the number of elements in the array. May include elements that are under construction from a concurrent `Push` call.
- `Read(i)`: read the element at index i ($0 \leq i <$ `Count()`).
- `Write(i, v)`: write v to entry at index i ($0 \leq i <$ `Count()`).
- `Push(v)`: Add an entry with value v to the end of the array.

Users expect the array to grow automatically to accommodate new elements added via `Push`. The textbook single-threaded implementation copies content into a new array of double the current size when growth is required. Unfortunately, this is very difficult to implement with latch-free programming. Current hardware atomic instructions only support limited-size updates (e.g, aligned 64-bit length variables), and cannot atomically copy arbitrary-sized arrays. The common solution of "shadowing", which copies the array separately and atomically updates the data structure reference to point to the new array, also does not work as it cannot prevent interleavings of the growth operation and concurrent read/write operations and can result in lost updates during growth. Programmers often need to settle for a significantly less performant alternative with a latch (e.g., protect normal array operations with a shared latch, and the growth operation with an exclusive latch), or invest in a more complex solution (e.g., a concurrent vector [1]).

This is a generalizable pattern. As shown in Fig. 1, almost-latch-free data structures can be broadly categorized as undergoing state transitions. In the REST state, the data structure is amenable to latch-free synchronization (e.g. using atomic instructions or other wait-free techniques); for example, resizable arrays are at rest when there is enough space left in its buffer. The data structure can also have multiple *disturbed* states, where stronger synchronization primitives such as latches are required. Resizable arrays, for example, have a singular disturbed state corresponding to array resizing, during which they cannot easily allow latch-free operations. While many data structures can be modeled in this fashion, almost-latch-free data structures are special in that they:

1. exist in the REST for the vast majority of the time,
2. only enter disturbed states in response to rare user requests or internal conditions,
3. eventually return to the rest state each time they enter a disturbed state, making disturbed states a transient anomaly rather than the norm.

To showcase how prevalent this paradigm is in modern data systems, we now present several examples from real systems

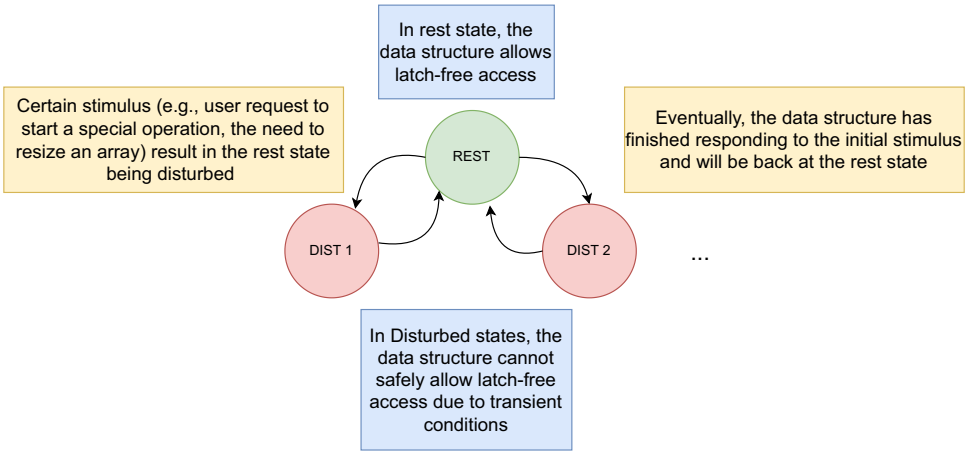


Fig. 1 Almost-latch-free archetype—We consider a data structure almost-latch-free if its operations can be classified into a common, latch-free rest state, and occasionally transitions to a disturbed state where latch-free access is no longer safe in response to a user request or some internal signal

that can be classified as almost-latch-free using the model we presented:

Metadata management Many distributed systems rely on cached metadata at each local machine for performance. For example, FASTER [26] distributes keys across multiple shards and maintains a mapping for key ownership locally at each shard; before an operation, the shard first verifies that it owns the key before operating on it. However, to implement this correctly is difficult: suppose that a shard decides to drop ownership of key k during an ownership change; it cannot declare the change done until all earlier validated requests for k finish; otherwise, a race can result in operations on k after the shard drops ownership. A naïve implementation may prevent this race by protecting the validation and the operation in a critical section using a shared latch, and updating ownership information under an exclusive latch, but can lead to scalability issues, especially if the underlying key-value store is more scalable than a shared latch [8]. We transform this problem into an almost-latch-free one by observing that metadata such as ownership information is largely static, and only modified infrequently compared to normal operations. We consider the ownership mapping at rest when it is static, and consider it to be in a disturbed state when a user requests to change the ownership mapping. Validation of the ownership mapping table without latch protection is then naturally safe during the rest phase as the mapping is assumed to be immutable.

Background data reorganization Hybrid Transactional Analytical Processing (HTAP) systems often need to transform cold, read-mostly data into a more read-optimized format. Such transformations are not always safe to complete with concurrent transactions; for example, in [30], the transformation process needs to compact each storage block such that tuples are laid out contiguously to each other, and then reorganize storage layout on the contiguous block for analytics. A correct implementation must exclude transactional access that may modify the block, without latching each block, which makes transactional access in normal situations more expensive. In the almost-latch-free model, a block is at rest if it is either hot or already transformed, and disturbed during transformation, which is infrequent compared to normal access.

Consistent snapshots Often, application builders maintain data structures in memory and periodically snapshot them to persistent storage for fault-tolerance. Such snapshots range from simply writing a counter to disk to checkpointing entire complex data stores such as [43], and often come with consistency requirements. Consider an example where we need to checkpoint a hash table that handles insert requests that need to be deduplicated—upon encountering a request, we first deduplicate the request by checking and updating a deduplication vector, and then handle the request. A consistent snapshot of this data structure means that any request logged in the deduplication vector has its effects reflected in the hash table. Then, the deduplication vector update and the hash table update must appear atomic with regards to the snapshot, which is difficult to achieve latch-free. In the almost-latch-free model, we consider the deduplicated hash table to be at rest when snapshotting is not under way, and only need to ensure atomicity of the updates when a snapshot is requested.

2.2 Epoch protection

Underlying the challenge of thread scalability is a fundamental hardware restriction—parallel threads run on physically distinct processor cores, and information must travel from one core to the other for thread coordination to occur. This incurs communication overhead that is often expensive for the application. Consequently, the key design principle for scalable multi-threaded data structures is to *avoid thread coordination where possible*. One natural design pattern to

develop based on this principle is to have threads run uncoordinated in predefined time periods called *epochs*, and only coordinate at epoch boundaries. We now introduce epoch protection as formulated in FASTER [8], which serves as a basis for our work.

At a high level, epoch protection consists of a global epoch number E and a set of participant threads T. Threads increment E using atomic instructions to signal the end of the old epoch. Participant threads each own a local copy of E, denoted $E_t, t \in T$, that periodically synchronizes with the global value. We define an epoch number e to be *safe* if $\forall t \in T, E_t > e$. Intuitively, an epoch number is safe when no thread is active with local epoch number e (although they may be running actively with a larger epoch number). Importantly, programmers often want to associate actions with the end of epochs. Executing an action after the associated epoch is safe implies all threads have discovered the intention to execute this action, and therefore presumably done the necessary preparatory work (e.g., finish using a resource) before releasing their epoch. We summarize the epoch framework API below:

- `Acquire()`:Add a thread t to T and set $E_t = E$.
- `Refresh()`: $E_t = E$, temporarily drops and immediately reacquires protection. If a thread is expected to be long-running, refresh is cheaper than calling `Release()` followed by `Acquire`.
- `Release()`: Remove current thread t from T.
- `BumpEpoch(e, action)`: increment E to e and associate `action` with the old epoch.

A code block is considered "protected" if it is guaranteed to execute within a single epoch, and therefore cannot interleave with epoch change or the associated action. Programmers can achieve protection by acquiring and not refreshing or releasing the epoch within a code block. We now give a concrete example in Fig. 2. Consider a simple workload where threads share access to a resource that must occasionally be renewed (e.g., a shared data structure that is moved out of a memory region for memory compaction). To renew a resource, we simply construct a new object for use and reclaim the old. It is safe to use the resource fully concurrently, but the resource must not be reclaimed when it is under active use. One might synchronize these threads by protecting resource usage with a shared latch, and renewing the resource under an exclusive latch. However, on most shared latch implementations, acquiring the shared latch requires a write operation to update a counter, which results in cache-line ping-ponging that limits thread scalability. Instead, we can use the epoch protection framework to solve this problem as shown in Fig. 2. Consider a thread t executing from line 6 while a separate renewal thread starts executing from

```
1  EpochFramework e;
2  Resource r;
3  ...
4  // Run concurrently on each thread:
5  e.Acquire();
6  while(true)
7  {
8    e.Refresh();
9    r.Use();
10 }
11 e.Release();
12 ...
13 // To update resource:
14 var old_r = r;
15 r = new Resource();
16 e.BumpEpoch(() => old_r.Reclaim());
```

Fig. 2 Example use case of epoch protection framework

line 14 in parallel. On line 16, the global epoch E is incremented from e_{old} to e_{new}. If t observed e_{new} when refreshing on line 8, then it must be using the new resource (which is not being reclaimed) as the read of r happened after line 15. If t observed e_{old} instead, it may see r before the renewal and use the old resource. However, because t does not refresh its epoch while using the resource, e_{old} does not become safe; by registering reclamation as an action to execute after the epoch is safe, we ensure reclamation cannot happen while the resource is in use. Compared to a shared latch, refreshing an epoch reads from the global shared epoch E, and a write to the local E_t variable, avoiding any potential cache-line ping-ponging on the common code paths.

3 Epoch-protected version schemes

The key challenge in implementing almost-latch-free data structures is to enable high common-case concurrency without introducing prohibitive implementation complexity due to rare disturbed states. One idea is to apply epoch protection as a performant synchronization primitive to this problem. The key missing ingredient here, however, is critical sections; even as the epoch protection framework ensures execution of `BumpEpoch` actions only after all participants have seen the end of the epoch, the action is still executed in parallel with other protected regions. This is sufficient for excluding some interleavings (e.g., free-before-use when garbage collecting), but is still fundamentally a latch-free technique. Imagine implementing a resizable array with epochs — if we were to similarly resize the array with a `BumpEpoch` call, the resize operation will occur in parallel with normal write operations, which still cannot protect against concurrent updates. This is a key limitation of epochs as formulated in [8], and one we address with our proposed framework of EPVS. In this section, we cover the interface and usage of EPVS, deferring the implementation details of EPVS for Sect. 4.

3.1 EPVS basics

EPVS simplifies almost-latch-free programming by providing a new abstraction called *versions*. At a high level, an almost-latch-free data structure starts a version change when it leaves the REST state, and completes the version change when it re-enters REST. Versions are explicitly numbered monotonically so they are distinguishable from each other, but version numbers are not necessarily always meaningful beyond this purpose. Developers protect code regions with epochs, so that each code region executes entirely within one state (rest or disturbed), and transitions between states by executing critical sections under mutual exclusions with all protected regions or other transitions. We summarize the (simplified for the moment) user API of EPVS below:

- `Enter()`: Begin executing protected region; protected region will execute entirely within the returned version.
- `Leave()`: Exit protected region.
- `Refresh()`: Equivalent to calling `Leave()` immediately followed by `Enter()`, but is more performant when called frequently.
- `AdvanceVersion(Action criticalSection, long targetVersion)`: Advance version to target version (or unconditonally to the next version if target version is -1) asynchronously, executing the supplied critical section under mutual exclusion for transition. Version advances monotonically, and each version can be advanced to at most once; otherwise, the call does nothing.

Intuitively, EPVS allows us to implement the aforementioned resizable array example by executing concurrent operations such as reads and writes in protected regions, and resizing the array as a version transition under mutual exclusion. We now walk through a pseudo-code implementation of this in Fig. 3. Users initialize EPVS instances as objects, much like with traditional latches. To protect a code region, users surround the protected region with `Enter` and `Leave`, as shown between lines 8–16 in Fig. 3; in protected regions, operations proceed latch-free. To push elements into the array, our implementation first uses an atomic counter (line 22) to obtain a unique position in the array, and then attempts to populate that spot under version protection. If the spot is beyond the limits of the current array, we execute the resize operation on line 31 as part of the state transition in an explicit `AdvanceVersion` call. Then, by the semantics of EPVS, the resize operation executes in mutual exclusion with all protected blocks, which ensures thread-safety. Threads then temporarily release version protection and wait until the submitted transition completes. For simplicity, our implementation spins (by re-entering the loop on line 23) until array growth is complete. Note here that `Enter` and `Refresh` calls explicitly return the version number, which EPVS will not advance until the corresponding `Leave` completes. For illustration purposes, we use this version number for deduplication purposes in this implementation. On line 32, we explicitly specify that EPVS should only advance to the immediate next version; because `AdvanceVersion` only allows for a version to be advanced to once, this ensures that when multiple concurrent threads are pushing onto a full array, only one request to resize is actually executed.

In this simple form, version schemes behave very much like traditional reader–writer latches, with concurrent operations protected under shared mode and version changes happening in exclusive mode. The difference here is that concurrent operations incur little to no overhead due to the use of epochs, until a version change happens; in exchange, version changes are more expensive than obtaining an exclusive latch, as we will show later in Sect. 6. This tradeoff, however, results in better overall performance as long as version changes are infrequent.

3.2 Advanced usage

While intuitive to understand, the simple version scheme we showed earlier limits concurrency in many cases. Specifically, if the critical section takes a long time (e.g., persisting to storage synchronously), all concurrent work is paused and scalability will suffer. The source of this issue is that we assumed it is unsafe to execute protected regions outside of the stable state. Consider again the example of resizable arrays; it is possible to define an intermediate, semi-stable state of COPYING and allow limited forms of operations while the array is resizing. We summarize the logic in Fig. 4, where REST is the stable state as before. In COPYING, both the old and the new arrays are visible, and some thread (either background or collaboratively by participants) is copying content from the old array to the new array. Let us assume we are resizing from size c to $2c$, copying from `arr_old` to `arr_new`. Firstly, we disallow writes in this state to avoid lost writes due to possible concurrent copying; writers retry until the array is back at REST. This implies that the array between index range $[0, c)$ is temporarily immutable in COPYING, and readers can read the value from `arr_old` instead. EPVS ensures that if a reader observes the array in COPYING (line 10), the array stays in COPYING due to epoch protection until it is dropped (line 17). As for push requests, any push requests serviced at this state must write to `arr_new` in the index range $[c, 2c)$, as the original array was full up to c. This region of `arr_new` will not be copied into from `arr_old` and is therefore safe to push into, and subsequently read from or written to normally. Now that we have shown how to operate the resizable array in RESIZE, we simply need to define how to transition between these two

```
1  class ResizableArray<T> {
2    EpochProtectedVersionScheme epvs;
3    T[] list;
4    int count;

6    ...

8    T Read(int index) {
9      epvs.Enter();
10     try {
11       // bounds check
12       ...
13       return list[index];
14     } finally {
15       epvs.Leave();
16     }
17   }

19   int Push(T value) {
20     var v = epvs.Enter();
21     try {
22       var pos = AtomicIncrement(count);
23       while (true) {
24         // Normal push into array tail
25         if (pos < list.Length) {
26           list[pos] = value;
27           return pos;
28         }

30         epvs.AdvanceVersion(
31           () => /* Resize */ ..., v + 1);
32         v = epvs.Refresh();
33       }
34     } finally {
35       epvs.Leave();
36     }
37   }
38   ...
39 }
```

Fig. 3 Pseudo-code for almost-latch-free resizable array with EPVS

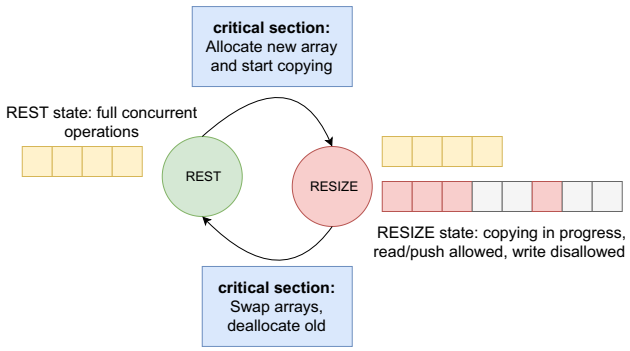


Fig. 4 Resizable array with two-phase state machine—adding an intermediate RESIZE state allows for concurrent read and push operations

states. With mechanisms similar to before, we can transition between these two states using critical sections. Intuitively, to start resizing, we allocate `arr_new` and start the copying process in the critical section from REST to RESIZE; to restore stable state after copying is done, we swing the array pointer from `arr_old` to `arr_new` and deallocate `arr_old`.

EPVS generalizes schemes like this into *transition state machines*. Transition state machines allow programmers to specify a number of semi-stable intermediate states like RESIZE during a version change, and move between these states using critical sections. Each protected region is guaranteed to execute entirely within one state that is made known at the start of execution. We show the EPVS state machine abstraction below:

- `GetNextStep(State current)`: Returns whether a next state is available and the state
- `OnEnteringState(State from, State to)`: Transition logic that executes in isolation before entering the next state.

Here, `State` is one 64-bit variable that encodes both a version number and a byte value for the phase of transition (e.g., RESIZE). A state machine always starts at REST in some version v, and ends at REST at a larger version, closely mirroring our state-based modeling of almost-latch-free data structures. Along the way, a state machine defines a series of semi-stable states, along with a critical section

that executes in isolation between two states. Clearly, this is a generalization of the API shown earlier, as the critical section is just a trivial state machine with one REST state that always transitions directly to itself. Importantly, a state machine can dynamically decide whether a next state is ready; for example, a resizable array may stay in RESIZE until the copying is complete, and only then will `GetNextStep` signal that the next state is REST for transition to occur. In summary, for advanced users, `Enter()/Refresh()` now return `State` as opposed to a single version number, and `AdvanceVersion` is complemented with `ExecuteStateMachine`, which similarly advances the version to a specified target, but takes a state machine instead of a critical section.

We now walk through the pseudo-code implementation of this two-phase resizable array in Figs. 5 and 6. In contrast to the earlier implementation, the 2-phase resizable array explicitly reserves references to both arrays during resizing. As described earlier, `Read` first acquires version protection, and then case on the observed phase to proceed differently, reading the corresponding underlying array. In the case of `Write`, because we disallow updates under COPYING, the implementation simply drops protection and retries until the array is at rest. Similarly, pushing into the array additionally cases on the current phase to decide which state to push into; the difference here is that instead of advancing the version with a critical section, we supply a state machine (Fig. 5) that resizes the array asynchronously (line 40 in Fig. 5). Note here that EPVS excludes interleavings where a resize is underway, but a concurrent push request is pushing into the old array (line 52 in Fig. 5, otherwise, a race may result in the copy into `list` not getting copied over into `newList`). This is because the transition that starts the copying executes in mutual exclusion with all protected blocks, which implies that no concurrent `Push` call is executing. Additionally, the transition starts only after `count` grows larger than `list.Length`, which, combined with the previous observation, implies that all `Push` calls into the old list has exited when copying starts.

4 Implementation

4.1 Efficient epoch framework implementation

We first describe FASTER's efficient, latch-free implementation[1] of the epoch protection framework sketched in Sect. 2, which serves as a basis for EPVS. At the core of this implementation is the global epoch table – an array of epoch entries consisting of a thread id, local epoch number, and padded to occupy an entire cache line to avoid false-sharing between threads. Threads access the epoch table as a latch-free linear probing hash table to add and remove themselves during `Acquire` and `Release` calls; for `Refresh`, they locally cache the array offset for their entry without a hash table lookup. To avoid repeatedly scanning the hash table, threads locally stash their index into the table in any instance of the epoch protection framework and reuse the same index for other instances for performance. The epoch protection framework additionally maintains a thread to index hash table for conflict resolution, ensuring that no other thread active in other instances has access to the same index. We allocate an array for epoch-action pairs called the drain list that `BumpEpoch` calls write to. Our implementation of the epoch framework is collaborative in nature; participants scan the epoch table at the end of each call to compute the safe epoch and traverse the drain list to perform any associated actions. Scanning the hash table, however, can be a relatively expensive operation if the number of threads is large, or if the memory access is across NUMA nodes. It is possible to reduce this overhead by offloading the scan responsibility to dedicated, affinitized threads, but we do not find it to be necessary in our experience for most workloads.

Note that in our current implementation, the global epoch table and drain list are statically preallocated and fixed-sized. When the number of concurrent threads is large compared to the size, hash table performance may degrade due to hash collisions, and eventually block new participants until some threads `Release`; similarly, frequent `BumpEpoch` calls may fill up the drain list and block until epochs are safe. We do not observe this to be a major concern in practical deployments, as it is often desirable to limit max threads in an application for performance anyways, and because our formulation of EPVS processes version changes sequentially, which limits the drain list size.

4.2 EPVS from epochs

We implement EPVS as a layer on top of the epoch framework. At a high level, threads enter and leave a version as they would an epoch, with the added responsibility to:

- block themselves when another thread is executing a critical section
- step the state machine if possible

Each EPVS instance holds an epoch framework instance, a single 64-bit atomic `State` struct, and a pointer to the current `StateMachine` (null when none is active). The `State` struct holds a version number, current phase, and a 1-bit flag on whether a thread is executing a critical section (intermediate state flag). When entering a version, a thread first acquires epoch protection and checks the current state, dropping epoch protection and retrying until the

[1] https://aka.ms/faster-epochs.

```
class ResizableArray<T>
{
  EpochProtectedVersionScheme epvs;
  T[] list, newList;
  int count;
  ...
}

class ArrayGrowthStateMachine<T>
  : StateMachine
{
  const byte COPYING = 1;
  ResizableArray<T> obj;
  bool copyDone = false;

  (bool, State) GetNextStep(State curr)
  {
    switch (curr.Phase)
    {
      case REST:
        next = new State(COPYING,
                    curr.Version);
        return (true, next);
      case COPYING:
        next = new State(REST,
                    curr.Version + 1);
        return (copyDone, next);
    }
  }

  void OnEnteringState(State from,
                       State to)
  {
    switch (from.Phase)
    {
      case REST:
        var newLen = obj.list.Length * 2;
        obj.newList = new T[newLen];
        // Execute asynchronously
        Task.Run(() => {
          Array.Copy(obj.list,
            obj.newList,
            obj.list.Length);
          copyDone = true;
        });
        break;
      case COPYING:
        obj.list = obj.newList;
        break;
    }
  }
}
```

Fig. 5 Psuedo-code for 2-phase resizable array EPVS state machine

state is not intermediate. To start a state machine (including when advancing with a critical section, which, as mentioned, is a trivial state machine), EPVS tries to atomically swap in the new `StateMachine`, retrying if there is another currently active state machine. In some cases, multiple threads may attempt to install state machines concurrently that do duplicate work (e.g., two threads inserting entries to an array concurrently request resizing at the same time). To prevent this, users may either handle it with application-specific logic or optionally specify the exact version a state machine is advancing to, and EPVS will disregard a state machine if it is behind the current EPVS version.

Stepping the state machine is achieved by:

1. calling `StateMachine.GetNextStep` to check if there is a next state available,
2. if so, attempting to atomically set state to the intermediate state
3. bumping the epoch with action to execute transition critical section, and transitioning from the intermedate state to next state

Because the transition critical section is associated with an epoch bump, all threads that observed the previous, non-intermediate state will have exited the epoch when the critical section executes, ensuring mutual exclusion. The next challenge in implementing EPVS is ensuring that the sys-

```
class ResizableArray<T>
{
  ...

  public T Read(int i) {
    var state = epvs.Enter();
    try {
      // bounds check
      ...
      if (state.Phase == COPYING
        && i < newList.Length)
        return newList[i];
      if (index < list.Length)
        return list[i];
      ...
    } finally {
      epvs.Leave();
    }
  }

  public void Write(int i, T v) {
    var s = epvs.Enter();
    try {
      // bounds check
      ...
      while (s.Phase == COPYING) {
        s = epvs.Refresh();
      }
      list[i] = v;
    } finally {
      epvs.Leave();
    }
  }

  public int Push(T v) {
    var state = epvs.Enter();
    try {
      var pos = AtomicIncrement(count);
      while (true) {
        switch(state.Phase) {
          case REST:
            if (pos >= list.Length) {
              epvs.ExecuteStateMachine(
                /* init state machine */,
                state.Version + 1);
              state = epvs.Refresh();
              continue;
            }
            list[pos] = v;
            break;
          case COPYING:
            ASSERT(pos >= list.Length)
            newList[pos] = v;
            break;
        }
      }
    } finally {
      epvs.Leave();
    }
  }
  ...
}
```

Fig. 6 Pseudo-code for 2-phase resizable array with EPVS

tem makes progress even when threads are not constantly refreshing. Imagine an otherwise quiescent resizable array implemented with EPVS, with one `Push` call triggering a resize. We would like the push call to succeed and the resizing to complete without the need for future calls. To guarantee this, we add an additional step attempt after each transition, which ensures that a state machine that does not delay steps (e.g., until copying is done in resizable array) can finish by itself without new threads entering the version. Otherwise, users are expected to explicitly invoke a step attempt when a new state becomes available.

```
1 b1:
2   x.AdvanceVersion(() => {
3       y.Enter();
4       ...
5       y.Leave();
6   }, ...);

8 b2:
9   y.AdvanceVersion(() => {
10      x.Enter();
11      ...
12      x.Leave();
13  }, ...);
```

Fig. 7 EPVS Deadlocks

4.3 Discussion

Just like any other concurrency building block, EPVS has a number of pitfalls and peculiarities that developers must pay attention to. We briefly outline some of them in this subsection:

Deadlocks It is possible for EPVS to deadlock much like with traditional latches. Consider a classical setup, as shown in Fig. 7, where two code blocks, b_1 and b_2 acquire protection from two EPVS instances, x and y. In an interleaving where both code blocks have only just entered the critical section, EPVS guarantees that no thread enters protected region while the version transition is underway; however, as seen on line 3 and 10, neither critical section will complete until it enters the protected region. There are two ways to handle this scenario, again analogous to handling deadlocks in traditional latches. First, programmers can *merge* the two EPVS instances into one, coarse-grained EPVS. This might appear to limit concurrency, but upon inspection is the preferred way to deal with deadlocks, as doing so hardly impacts thread scalability on the common path thanks to epoch protection, and the almost-latch-free assumption states that version transitions are rare, and therefore unlikely to have a major impact on scalability. Second, programmers can apply programming discipline to avoid wait-for cycles, either by rewriting the program to not enter protected region within transitions, or by rewriting b_2 to advance x first, and execute the y version change as part of the critical section. Note that simply entering protected region outside of the `AdvanceVersion` call does not work, as the critical section is asynchronous and may execute on another thread.

Advancing under protection A common pattern when using EPVS is to acquire protection, check some condition, and then decide to advance the version (e.g., line 30 in Fig. 3). This may lead to complications, as the version change cannot take place until the code block exits and leaves the current version, which temporarily creates a self-dependency for progress. Recall also that even though EPVS advances versions asynchronously, it may need to continuously retry (e.g., due to an existing state machine already underway) before it can register or discard a request to advance the version. Then, consider a situation where a call to advance the version is continuously retried under protection due to another active state machine; the retry continues until the other state machine completes, and the other state machine cannot complete until the retry finishes, and the code block exits from protection. To avoid situations like this, we have written the `AdvanceVersion` call to allow *spurious failures*, and return without registering a state machine, even when future retries may succeed. Like line 33 of Fig. 3, users should retry `AdvanceVersion` calls only after exiting or refreshing protection.

Epoch sharing EPVS is a thin layer of state machine logic on top of the epoch protection framework. This means that multiple data structure can, in theory, share the same underlying epoch framework instance. Because instantiating an epoch framework is more expensive than traditional latches (more memory space required for the epoch table and drain list), sharing epochs can significantly reduce this overhead, especially when there are many EPVS instances. Applications can either do so explicitly by using the same EPVS instance and install their own state machines, or each have its own EPVS instance sharing an underlying epoch table. Doing so comes with some disadvantages, however. Generally, as the number of EPVS instances multiplexed increases, the epoch framework will, on average, bump its epoch more frequently at the cost of performance. Additionally, depending on the implementation, sharing an EPVS instance forces version changes to be serialized even when they may be from different data structures, introducing performance interference, whereas sharing an underlying epoch must support re-entrance to avoid deadlocking when multiplexed instances interleave.We leave a more detailed study of the trade-offs and workarounds for future work.

Performance inconsistencies It is worth pointing out that EPVS is, as one might expect, *almost* latch-free, rather than latch-free. Pedandically, this means that threads syn-

chronizing using EPVS potentially blockingly wait for a version transition, nullifying the wait-free, non-blocking aspect of the original epoch framework when a version change is underway. Practically, this translates into performance instability, as EPVS blocks and take longer to acquire protection/allow useful work while in disturbed states. It is also not uncommon for almost-latch-free data structures to naturally operate at reduced capacity during disturbed states (e.g., FASTER is forced to copy records instead of updating them in-place during checkpointing, drastically reducing maximum write throughput until checkpointing is complete). Recent work in distributed systems [] proposed the idea of metastable failures, where a system optimized for the common path can enter a state of sustained failure (metastable) after a trigger moves its workload off the optimized path and temporary performance degradation is sustained by work amplification or decreased efficiency. EPVS appears particularly vulnerable to metastable failures (more specificially, a single-node variant of it) as it is built to optimize for the common stable state and has degraded performance in vulnerable states. It is worth noting, however, it is *not* a good idea to eliminate the possibility of metastable failures by not using techniques like EPVS– there is fundamental trade-off between efficiency and safety in the form of performance or capacity redundancy, and real-world resource constraints often force developers to pursue the former. That said, EPVS programmers must take special care to handle such scenarios via request throttling or other mechanisms to avoid staying in the metastable state for prolonged periods of time.

5 FASTER case study

We have used EPVS to simplify synchronization logic in FASTER and many of its derivative systems [24, 31, 32]. In this section, we provide two examples of EPVS usage in the FASTER system. We first show how we use EPVS to implement the aforementioned key ownership management scheme in the distributed version of FASTER; then, we show how EPVS can be used to simplify FASTER's asynchronous checkpointing logic, as described in [43].

5.1 Cluster ownership management

As discussed in Sect. 2, the distributed version of FASTER relies on a key ownership map at each shard to ensure that only owners operate on a key. We walk through a simplified implementation of this logic in Fig. 8, using a ownership tracking scheme similar to Redis-cluster with 16-bit hash buckets. Each FASTER shard owns a single-threaded set data structure holding hash bucket ids, and relies on EPVS for both thread-safety at the data structure level, and to ensure that operations are not executed on the wrong shard or become lost during ownership changes. When a normal operation arrives, the FASTER shard first acquires protection on line 6, and then proceeds to use a simple membership check to determine if it is allowed to execute the operation; protection extends until the operation is finished. When the server encounters a user request to acquire/relinquish ownership of a bucket, it instead executes the logic as a version transition (lines 24 and 30). Thread-safety is guaranteed for the ownership set because EPVS guarantees that it is never modified when normal requests read into it under protection, and that updates to it are serialized as version transitions. For correctness during ownership changes, FASTER models an ownership transfer as a relinquish followed by an acquire, with a transient period of migration when no shard owns a bucket. On line 28, we first initiate a version transition to drop the bucket, and wait for it to complete on line 29. Then, as we exit from the loop, EPVS ensures that all future requests will see the ownership mapping without the relinquished bucket, and it will not be modified; it is therefore safe to begin the migration logic at that point.

5.2 Asynchronous checkpointing

Traditional DBMS recovery methods rely on the write-ahead log, which is a serial bottleneck, especially for update-intensive workloads. To alleviate such bottlenecks, FASTER utilizes the *concurrent prefix recovery* (CPR) model to asynchronously and incrementally checkpoint DBMS state. With the CPR scheme, FASTER threads modify entries in-place in the common code path, but temporarily freeze them during checkpoints; at this time, threads switch to a special read-copy-update mode while a background thread flushes the DBMS state to storage. CPR guarantees that the checkpointed content constitutes a consistent cut across all threads. FASTER uses a checkpoint state machine to model this process and identify checkpoints using sequential version numbers, much like EPVS's approach, as shown in Fig. 9. We describe each phase on a high level as follows:

- **Rest Phase:** In the rest phase, FASTER exists in a stable state and processes requests normally by updating records in-place.
- **Prepare Phase:** Requests processed in the prepare phase are committed as part of the current checkpoint; threads may advance to the next phase voluntarily or as a result of consistency issues (e.g., reading a record written by another thread in the next phase, which does not belong to the commit).
- **In-Progress Phase:** Threads cross the CPR cut in the in-progress phase – requests accepted in this phase or later do not belong to the current checkpoint.
- **Wait-Flush Phase:** Records of the current checkpoint are flushed asynchronously to storage.

```
class FasterShard {
  EpochProtectedVersionScheme epvs;
  HashSet<BucketId> ownership;
  ...
  void ProcessRequest(Request r) {
    epvs.Enter();
    try {
      // Safe to use non-thread-safe data structure
      // here because ownership is read-only when
      // under protection
      if (!ownership.Contains(r.Bucket))
        // Report Error
        ...
      ExecuteRequest(r);
    } finally {
      epvs.Leave();
    }
  }
  ...

  void AcquireOwnership(BucketId b) {
    // Migration logic not shown
    ...
    epvs.AdvanceVersion(() => ownership.Add(b), -1);
  }

  void RelinquishOwnership(BucketId b) {
    epvs.AdvanceVersion(() => ownership.Remove(b), -1);
    while (/* version not advanced */...) {}
    // Migration logic not shown
    ...
  }
}
```

Fig. 8 Using EPVS to implement ownership mapping in FASTER

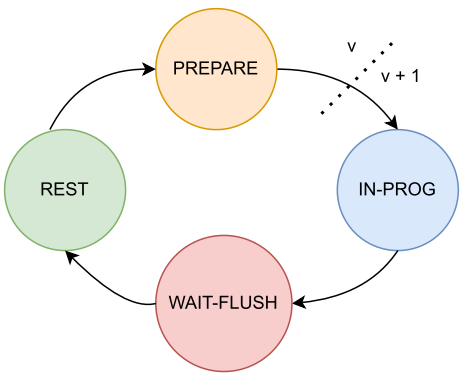


Fig. 9 The original FASTER CPR state machine

To implement this scheme correctly on FASTER's epoch framework requires much work, however, due to the lack of mutual exclusion. FASTER implemented a *marker* subsystem on top of the epoch framework, which adds a global version and phase variable packed into a single 64-bit integer, and a vector of 64-bit integers in each thread's local storage. Each phase of FASTER's state machine is mapped to an index on the vector, and developers manually note down the version number of a thread at the offset when the thread reaches a phase. A thread advances its phase at epoch boundaries, if either the global phase is ahead its local phase, or it scans other active threads and discovers that all of them have arrived at the same phase, at which point it advances the global phase to the next phase. The marker system is highly complex as it is completely latch-free – other threads may be actively executing requests when a phase advances, and an inactive thread rejoining the system may start out arbitrarily behind the current phase in its phase vector, leading to many subtle races and bugs.

We now briefly sketch out an implementation of the CPR state machine in EPVS. Unlike the original CPR state machine, our new implementation just uses two states, a rest state and the wait-flush state, as shown in Fig. 10. We determine the checkpoint content (i.e., the entries to persist) in the transition critical section to wait-flush. Threads in wait-flush read-copy-update, and transition the system back to rest when flush is complete. Note that EPVS trades-off some concurrency for simplicity here. For example, threads using EPVS protection cannot start in the next phase until all threads exit the previous phase, whereas with epoch and marks, some threads may continue executing in the previous phase until they refresh protection even after the rest of the system moves on. Consequently, the original FASTER CPR machine features an additional prepare phase, which ensures that no thread operates without knowledge of the checkpoint (i.e., in rest) when the checkpoint logic is executed (i.e., the system enters in-progress); this is not necessary in the EPVS version at the expense of threads potentially delaying longer

```
class FasterEpvsStateMachine : StateMachine {
  FasterKV faster;
  ...

  override bool GetNextStep(State curr, State next) {
    switch(curr.Phase) {
      case REST:
        next = new State(WAIT_FLUSH), curr.Version)
        return true;
      case WAIT_FLUSH:
        next = new State(REST,curr.Version + 1);
        // Return based on whether the flush is complete
        return ...;
    }
  }

  override void OnEnteringState(State from,
                                State to) {
    switch(curr.Phase) {
      case REST:
        // Initialize and start checkpoint by capturing
        // a snapshot of FASTER-KV hybrid log in the form
        // of an offset
        ...
        break;
      case WAIT_FLUSH:
        // Complete checkpoint by resetting local data
        // structures and persisting metadata
        ...
        break;
    }
  }
}

class FasterKV {
  EpochProtectedVersionScheme epvs;
  ...
  Status Op(...) {
    var state = epvs.Enter();
    // original FASTER logic for the operation
    ...
    epvs.Leave();
  }
  ...
  bool TakeHybridLogCheckpoint(...) {
    return epvs.ExecuteStateMachine(
      new FasterEpvsStateMachine(this));
  }
}
```

Fig. 10 Psuedo-code for FASTER checkpoints using EPVS

before starting the next phase. As we show later in Sect. 6, this is not a high price to pay except under extreme conditions, such as when a straggler thread does not refresh for a copious amount of time, halting progress for the rest of the system.

6 Evaluation

We now evaluate the performance of EPVS in a variety of workloads, seeking to address the following questions:

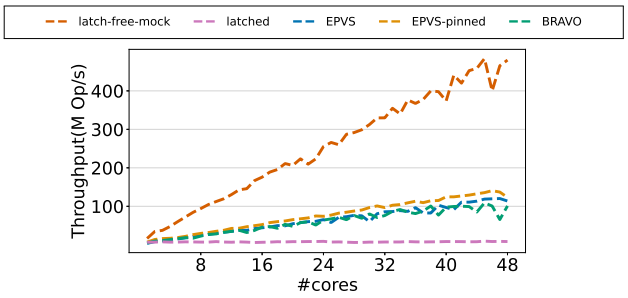
- Does EPVS generalize to a variety of data management workloads?
- Does EPVS provide increased scalability and lower cost compared to alternative synchronization solutions?
- Does EPVS perform gracefully under extreme conditions (e.g., frequent version changes, frequent thread context switches, limited memory space)?

We implement EPVS in C# as discussed before. We ran all experiments on the Azure public cloud [9], using the machine type Standard_D48s_v3, which has 48 vCPUs (2.60 GHz Intel Xeon Platinum 8171 M CPUs) within one CPU socket. We also implement the BRAVO algorithm [11] in C# according to the paper as a comparison baseline. BRAVO uses a globally shared *visible readers table* to record presence of readers, and instance-local bias bits to denote the preferred access mode, which corresponds to the epoch table and per-

instance phase indicator of EPVS. Both EPVS and BRAVO runs with a hash table size of 4096 entries unless otherwise specified.

6.1 End-to-end experiments

Resizable array We now compare throughput of 6 implementations of our resizable array example in earlier sections. Here, latch-free-mock is an ideal (and unrealistic) latch-free implementation where we provision a large array such that no resizing is required for the experiment. Note that this is not a reasonable implementation for most applications and instead represents an upper limit for performance.We provide an additional baseline where we protect array resizing under exclusive latches, and other operations under shared latches. For EPVS, we implement the two schemes detailed in Sect. 3, as simple-EPVS and 2-phase-EPVS, respectively. We also supply an additional implementation of simple-EPVS where threads are long-running and refresh, rather than leaving and rejoining epoch for protection (EPVS-pinned). We randomly generate one million operations with $p\%$ push operations, and $(1-p)/2\%$ each for read and write operations. We first study the performance of the array without resize in Fig. 11. Here, we start the array at 1 million elements and issue 1 million operations per threads, with 50% reads and writes. We can see that EPVS-based solutions are linearly scalable, similar to the latch-free baseline, and perform significantly better than the latched alternatives. EPVS and BRAVO are roughly equal in performance normally, but EPVS when pinned clearly outperforms alternatives, because threads no longer go through the shared hash table in this code path. That said, overall throughput using EPVS is quite a bit lower than the latch-free-mock due to the intrinsic overhead of epochs (a few random memory accesses), which is expensive compared to the extremely fast array operations (one random memory access). As we show in other experiments, this cost is negligible for many other workloads. We then study performance of our scheme with varying percentage of push workloads, starting from an empty array to force more resizing. As the results in Fig. 12 show, push itself cannot be a linearly scalable operation as push percentage increases, as there is workload-induced contention at the tail of the array. Although the two-phase-EPVS array implementation is in theory superior to the simple-EPVS one, displays no significant advantage over simple-EPVS when push percentage is low. In practice, we have discovered that additional phases in the state machine is usually only worth the overhead if protected operations would otherwise have to wait for I/O or other similarly expensive process. To illustrate this, we tweak the benchmark to add a 10ms delay to each array resize. Then, as shown in Fig. 13, the 2-phase solution clearly outperforms the simple baseline.Lastly, we demonstrate that EPVS performs better than BRAVO under adverse circumstances. In



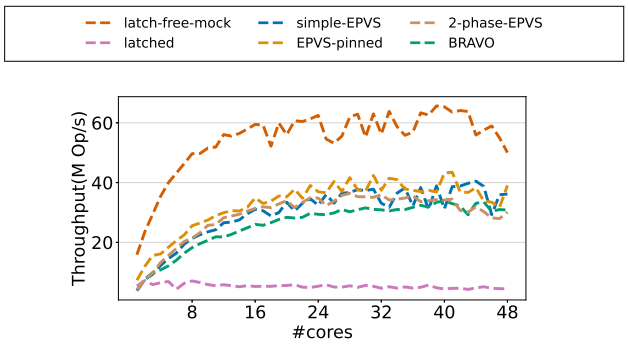
(a) p=0%

Fig. 11 Scalability of EPVS resizable array—read/write

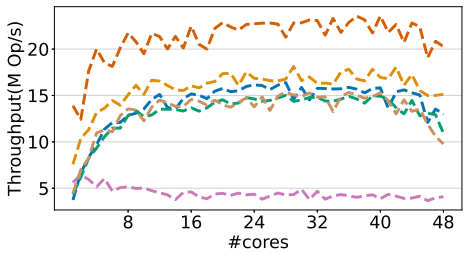
Fig. 14, we have reduced the hash table size to 128 for both EPVS anf BRAVO to increase contention and collision. As the esults show, EPVS is clearly more robust than BRAVO at handling this – this is because EPVS's epoch table has a linear probing design to resolve conflicts, whereas BRAVO falls back to the slowpath on such conflicts. EPVS is also able to avoid slowdown from small epoch tables by pinning threads to epochs thanks to its epoch-based design.

Cluster ownership management Figure 15 shows the performance of our prototype of a sharded version of the FASTER key-value store described before. We use a YCSB workload with 50:50 read-write ratio and uniform distribution, and report the average throughput on three configurations: one without any validation, one with validation protected by a reader–writer latch, and one with EPVS. We do not trigger any ownership transfer for the purpose of this benchmark. We can see that EPVS has similar scalability as the no-validation baseline, and is much more scalable than the naive latch-based baseline.

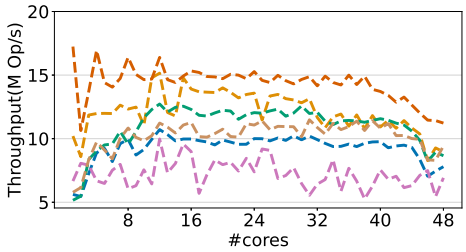
Asynchronous checkpointing We now compare EPVS against a hand-written latch-free epoch-based checkpointing solution as described in [43]. We again use a YCSB workload with 50:50 read-write ratio and uniform distribution, triggering checkpoints periodically. Checkpoints are written to `/dev/null/` for speed, such that the checkpointing mechanism itself, rather then the disk, is stressed. Note that we only show scalability on a single CPU socket to minimize interference from NUMA in this experiment. We use two models of operations: fine-grained, where each protected region consists of one key-value operation, and coarse-grained, where each protected region consists of 16,384 operations. As shown in Fig. 16, EPVS retains most of the performance and scalability of the original FASTER CPR implementation when operations are fine-grained and checkpoints are frequent. It is important to note that the original CPR algorithm was designed in such way that threads never block each other unless accessing the same record during checkpointing, from which most of its complexity derives. The EPVS-based solution, on the other hand, forces threads to wait while a



(a) p=10%

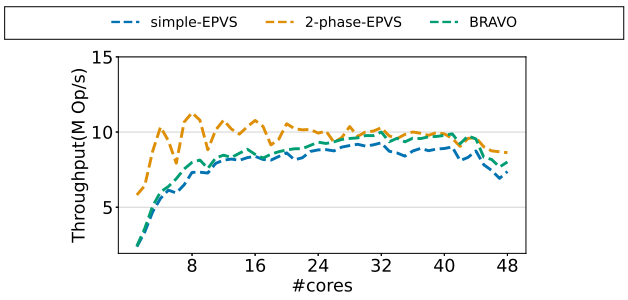


(b) p=50%



(c) p=100%

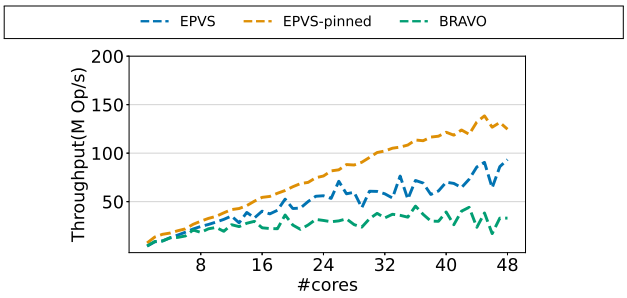
Fig. 12 Scalability of EPVS resizable array



(a) p=100%

Fig. 13 Scalability of EPVS resizable array—with resize delay

critical section is underway. We can see this manifest in the coarse-grained operations case in Fig. 16, where the original solution outperforms EPVS. Such cases are rarely encountered in practice, however. In exchange for performance in this edge case, we were able to reimplement the necessary checkpointing mechanism in FASTER using EPVS in just



(a) p=0%

Fig. 14 Scalability of EPVS resizable array—with constrained hash table size

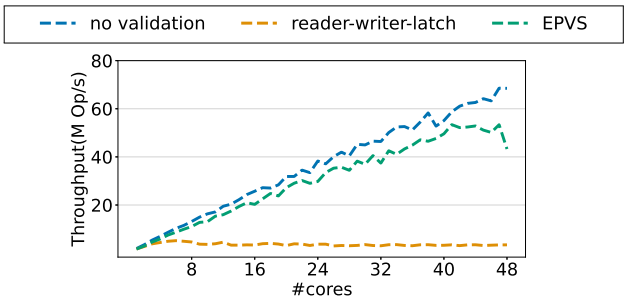


Fig. 15 Scalability of FASTER-remote key validation

a few hours with dozens of lines of code, whereas the original solution took months of subtle engineering and many lines of code with supporting mechanisms. While EPVS is not optimal in performance, it achieves most of the speed-up of FASTER CPR with a fraction of its complexity.

6.2 Microbenchmarks

We now evaluate the performance of EPVS in a series of microbenchmarks to better understand its limits and sensitivity to implementation parameters. For these experiments, each thread protects hashing of some local bytes, which simulates work but does not generate any contention between threads; the synchronization method used is the only source of contention. For each experiment, we run 1 million simulated operations per thread.

Impact of version change frequency In this experiment, we vary the frequency of version change by participating threads. Each thread will trigger a version change with probability p for each operation. Other than EPVS, we also provide two baselines where operations are executed without any synchronization (latch-free) or with mutual exclusion (latched). We can see from the results in Fig. 17 that EPVS retains competitive performance when version change is infrequent as before; however, performance begins to degrade for more frequent version changes, both because more synchronization is required and because version change is expensive.

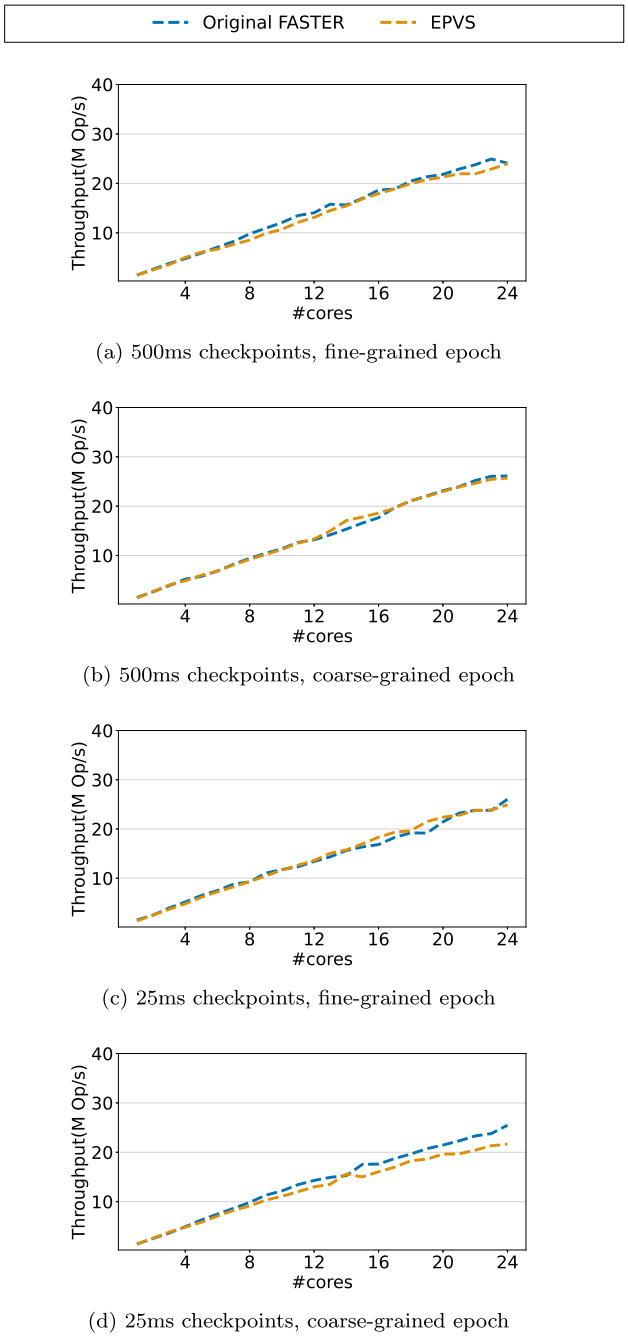


Fig. 16 Scalability of FASTER-KV checkpointing

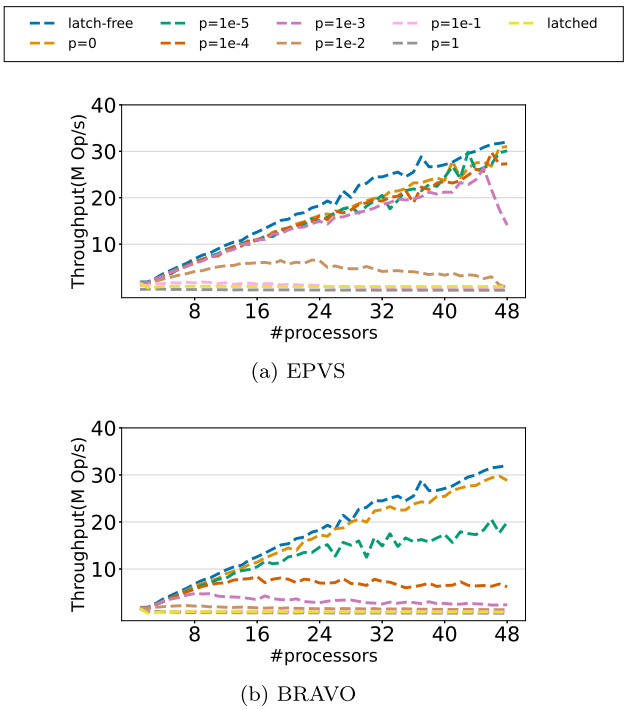
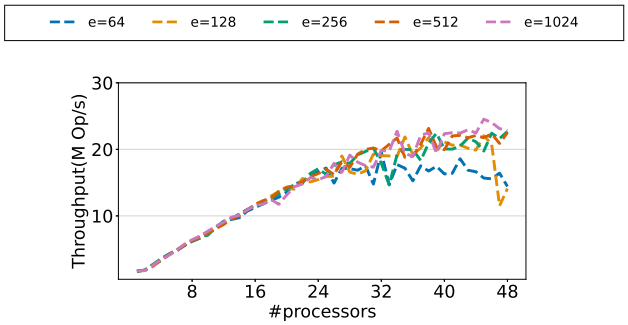


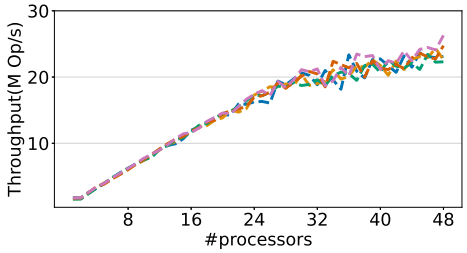
Fig. 17 Sensitivity analysis of version change frequency

When version change is frequent, EPVS is less performant than simply executing protected regions under a latch. This is because version change is relatively expensive compared to a latch given epoch and state machine mechanics. To summarize, EPVS performs best when version change is rare compared to normal operations. We conduct the same experiment for BRAVO and observe that BRAVO is much more sensitive to frequent exclusive acccess, likely because it is designed to be more conservative than EPVS in bypassing protection in the common case. Noticably, however, BRAVO performs no worse than a latch in the worst case of $p = 1$, unlike EPVS, thanks to this design philosophy.

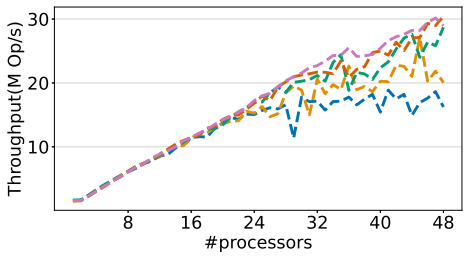
Impact of hash table size Recall from Sect. 4 that EPVS is implemented on top of the epoch protection framework, which is in turn built on a fixed-size latch-free hash table. In this experiment, we vary the size of this table and report the resulting throughput. All experiments execute with version change probability 1e−4. We can see from the results on the left in Fig. 18 that with a small epoch table size, EPVS experiences reduced scalability as threads crowd the table and EPVS operation repeatedly scans the table for a spare slot. This problem is alleviated as table size increases; we have found that in general, provisioning a table with at least double the thread count is desirable for performance. One would expect that with large tables, computing the safe epoch becomes more expensive as it requires scanning every entry in the table. This effect does not appear to be profound, as the epoch table is still a relatively small chunk of contiguous memory even for large table sizes, which makes it cheap to scan. Recall from earlier as well that we can optimize EPVS to skip hash table lookup for long-running threads, by using `Refresh` instead. We show such a run on the right side of Fig. 18, and see that indeed this improves scalability for smaller table sizes. Lastly, we perform the same experiment for BRAVO and observe that BRAVO is more sensitive to smaller hash tables, and only performs well when the table



(a) enter-leave



(b) refresh



(c) BRAVO

Fig. 18 Sensitivity analysis of epoch table size

is much larger than the number of threads. This is because a hash collision in BRAVO results in an overly conservative switch to the slow path, whereas EPVS handles collisions via linear probing.

Impact of phases To study the overhead of a complex state machine over a coarse-grained critical section, we fix the total amount of work (measured in iterations of hashing) in a version change and vary the number of phases it is split across. As reported in Fig. 19, there is little overhead introduced by additional stages. This makes sense because phases and versions are implemented based on the same epoch protection primitives, and therefore the only overhead the state machine adds is logic that looks up the next transition, which is lightweight. However, users should still be aware, as we showed earlier in the resizable array example, that additional phases can result in added application complexity, which may manifest as an overall loss, even when performance overhead is minimal on the EPVS level.

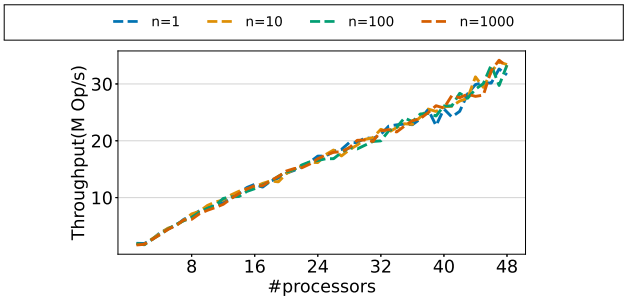


Fig. 19 Impact of number of phases on scalability

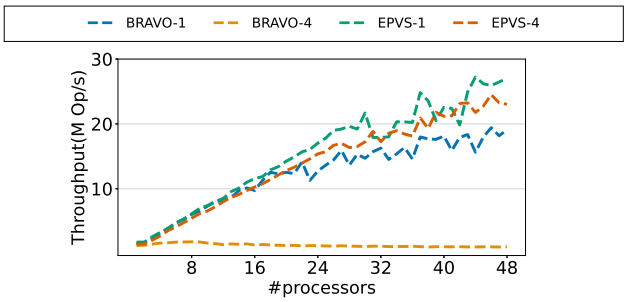


Fig. 20 Nesting

Impact of nesting We conduct a simple experiment of nested protection, where a protected block acquires protection sequentially before proceeding, and releasing them in reverse order. In this experiment we show results of 1 acquisition (i.e. no nesting) and 4 nested acquisitions, under the same microbenchmark with no version changes. As reported in Fig. 19, EPVS sees very little change in performance, whereas BRAVO sees significant performance degradation during nesting. This is due to a flaw in the BRAVO design, where different lock instances share one underlying hash table with no collision handling scheme other than reverting to the slow path. Because BRAVO uses thread identity as a key to its hash table, nested protection *guarantees* such degenerate case. To avoid this issue, BRAVO will need to either introduce collision or use a different scheme for hashing (e.g. hashing with a combination of thread identity and lock instance id).

Impact of sharing Lastly, we briefly study the potential impact of epoch sharing. As discussed earlier, sharing is one way to reduce complexity in deadlock detection and memory overhead by protecting many data structures with one coarse-grained latch primitive. In this experiment, we provision 8 instances of EPVS, BRAVO, and the C# reader–writer latch. In the no-sharing scenario, each thread is randomly assigned one instance (maximum 6 threads sharing an instance) for use in the same microbenchmark, whereas in the sharing scenario, every thread uses the same instance. We run a workload with relatively frequent version change ($p = 1e{-}4$)

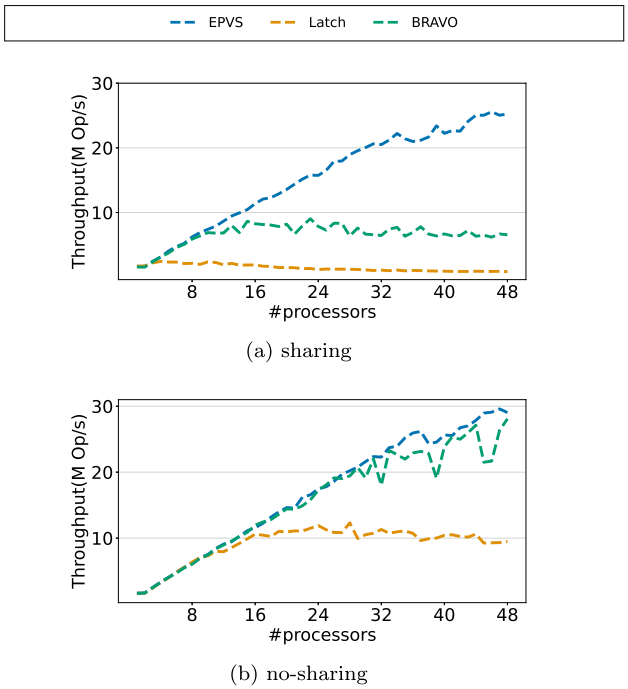


Fig. 21 Epoch sharing

and observe that EPVS is able to retain most of its scalability, whereas BRAVO and the C# reader–write latch sees noticeable drop in scalability.

7 Related work

Traditional latches Latches, in various forms, are the most common synchronization primitives for protecting shared resources in concurrent programming [21]. Reader–writer latches, or shared-exclusive latches, are the most related to EPVS, allowing arbitrary numbers of shared readers or one exclusive writer access at any given time. Traditional reader–writer latch designs [6, 40, 47] use a shared counter to record presence of readers, which becomes a bottleneck when there are many concurrent readers; this is because concurrent readers frequently update the shared counter, which introduces frequent cache invalidation [10, 12]. To alleviate this bottleneck, prior work has proposed distributing the shared counter across threads at various granularity [7, 25, 28, 33] at the expense of increased memory footprint. EPVS is closest to the concept of *biased locking* [42, 45, 51], where certain threads are allowed bypass the normal, expensive codepath when acquiring and releasing the latch, and other threads must revoke this bias via a heavy codepath. Other work explores the possibility of achieving a similar effect through hardware-based lock elision [36, 44]. In this line of work, BRAVO [11] is of particular note because it applies the concept of biased locking to reader–writer latches, leading to a user experience very similar to EPVS. Apart from the similarity in implementation, however, EPVS is still fundamentally an epoch-based mechanism, and is interoperable with pure epoch-protected code such as FASTER. BRAVO is, in contrast, a latch-based approach that regresses to an underlying reader–writer latch in the case of hash collisions or recent write access. EPVS, on the other hand, has no such fallback mechanism and has extra logic (e.g., reassigning a thread to a different slot in the hash table) to ensure correctness in these situations.

Latch-free programming and epoch protection While latches are often easier to reason about and use in most cases, they limit concurrency and introduce bottlenecks in many highly concurrent workloads. There is much work in building data structures with operations that do not require mutual exclusion, i.e. latch-free through the use of hardware-level atomic instructions like compare-and-swap [5, 18, 38, 39, 50, 53]. However, such latch-free data structures are not always faster than their latch-based counterparts, due to reasons such as failed and retried non-blocking operations and additional data copying [4]. Additionally, latch-free programming suffers from additional complexity in memory reclamation and other background maintenance tasks [22], which has lead to the introduction of epochs [27] as an efficient garbage collection mechanism for concurrent binary search trees. Epochs have continued and continue to support garbage collection in more modern concurrent data structures such as the Bw-tree [29, 37, 52], but epochs have been used in a more general sense as well. Silo [49] is an OLTP system that uses epochs to improve transaction throughput, where threads commit transactions only at the end of epochs to reduce the amount of synchronization overhead. Similarly, the concept has been extended to later transactional systems such as [35, 48, 54]. EPVS follows a more general notion of epoch protection for thread coordination and safe code scheduling, as formulated in FASTER [8].

Transactional memory Much of the complexity in latch-free programming arises from the limited size of the atomic unit (e.g., 64-bit compare-and-swap), which has lead to proposals of multi-operation atomicity support in the form of transactional memory [34]. The original proposal [20] proposes an extension to existing multiprocessor cache-coherence protocols on the hardware level; later work [46] extends this idea with software transactional memory. While the initial transactional memory proposals are latch-free and obstruction-free [17, 19, 41], the community eventually converged on latch-based methods for performance and simplicity [13–15]. EPVS, although not transactional, follows a similar philosophy, replacing a previously non-blocking solution with a potentially blocking one for simplicity, and may benefit from similar improvements and techniques from

the transactional memory community. We leave this discussion for future work.

8 Conclusion

We presented EPVS, a framework for concurrent programming that combines the raw performance of latch-free programming with the intuitive guarantees of critical sections and mutual exclusion. EPVS is implemented on top of an efficient epoch protection framework that can easily scale up to millions of fine-grained operations per second and dozens of cores. Our evaluation of EPVS suggests that it is both easy to use and highly efficient. EPVS is available in open source as part of Microsoft's FASTER project [2, 3].

Funding Open Access funding provided by the MIT Libraries.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Intel®oneAPI Threading Building Blocks (oneTBB) Documentation for concurrent_vector. https://www.intel.com/content/www/us/en/develop/documentation/onetbb-do (2021)
2. Epoch Protected Version Scheme (source code). https://aka.ms/epvs (2022)
3. Microsoft FASTER. https://github.com/microsoft/FASTER (2022)
4. Alemany, J., Felten, E.W.: Performance issues in non-blocking synchronization on shared-memory multiprocessors. In: Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing, PODC'92, New York, NY, USA, pp. 125–134. Association for Computing Machinery (1992)
5. Bershad, B.N.: Practical considerations for lock-free concurrent objects. 9 (2000)
6. Brandenburg, B.B., Anderson, J.H.: Spin-based reader-writer synchronization for multiprocessor real-time systems. Real-Time Syst. **46**(1), 25–87 (2010)
7. Calciu, I., Dice, D., Lev, Y., Luchangco, V., Marathe, V.J., Shavit, N.: Numa-aware reader–writer locks. SIGPLAN Not. **48**(8), 157–166 (2013)
8. Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., Barnett, M.: FASTER: a concurrent key-value store with in-place updates. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD'18. ACM (2018)
9. Copeland, M., Soh, J., Puca, A., Manning, M., Gollob, D.: Microsoft Azure: Planning, Deploying, and Managing Your Data Center in the Cloud, 1st edn. Apress, USA (2015)
10. Dice, D., Hendler, D., Mirsky, I.: Lightweight contention management for efficient compare-and-swap operations. In Wolf, F., Mohr, B., Mey, D. (eds.) Euro-Par 2013 Parallel Processing—19th International Conference, Aachen, Germany, August 26–30, 2013. Proceedings, volume 8097 of Lecture Notes in Computer Science, pp. 595–606. Springer, Berlin (2013)
11. Dice, D., Kogan, A.: BRAVO—Biased locking for reader–writer locks. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19), Renton, WA, July 2019, pp. 315–328. USENIX Association (2019)
12. Dice, D., Lev, Y., Moir, M.: Scalable statistics counters. In: Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'13, New York, NY, USA, pp. 43–52. Association for Computing Machinery (2013)
13. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) Distributed Computing, pp. 194–208. Springer, Berlin (2006)
14. Dice, D., Shavit, N.: Understanding tradeoffs in software transactional memory. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO'07, USA, pp. 21–33. IEEE Computer Society (2007)
15. Dice, D., Shavit, N.: TLRW: return of the read-write lock. In: Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'10, New York, NY, USA, pp. 284–293. Association for Computing Machinery (2010)
16. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD'08, New York, NY, USA, pp. 981–992. Association for Computing Machinery (2008)
17. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA'03, New York, NY, USA, pp. 388–402. Association for Computing Machinery (2003)
18. Herlihy, M.: A methodology for implementing highly concurrent data structures. In: Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP'90, New York, NY, USA, pp. 197–206. Association for Computing Machinery (1990)
19. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, PODC'03, New York, NY, USA, pp. 92–101. Association for Computing Machinery (2003)
20. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA'93, New York, NY, USA, pp. 289–300. Association for Computing Machinery (1993)
21. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, 1st edn. Revised Reprint. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2012)
22. Herlihy, M.P., Moss, J.E.B.: Lock-free garbage collection for multiprocessors. In: Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 229–236 (1991)
23. Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., Falsafi, B.: Shore-MT: a scalable storage manager for the multicore era. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT'09, New York, NY, USA, pp. 24–35. Association for Computing Machinery (2009)

24. Kanellis, K., Chandramouli, B., Venkataraman, S.: F2: designing a key-value store for large skewed workloads (2023)
25. Kashyap, S., Min, C., Kim, T.: Scalable NUMA-aware blocking synchronization primitives. In: Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC'17, USA, pp. 603–615. USENIX Association (2017)
26. Kulkarni, C., Chandramouli, B., Stutsman, R.: Achieving high throughput and elasticity in a larger-than-memory store. Proc. VLDB Endow. **14**(8), 1427–1440 (2021)
27. Kung, H.T., Lehman, P.L.: Concurrent manipulation of binary search trees. ACM Trans. Database Syst. **5**(3), 354–382 (1980)
28. Lev, Y., Luchangco, V., Olszewski, M.: Scalable reader–writer locks. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'09, New York, NY, USA, pp. 101–110. Association for Computing Machinery (2009)
29. Levandoski, J.J., Lomet, D.B., Sengupta, S.: The BW-tree: a B-tree for new hardware platforms. In: Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013), ICDE'13, USA, pp. 302–313. IEEE Computer Society (2013)
30. Li, T., Butrovich, M., Ngom, A., Lim, W.S., McKinney, W., Pavlo, A.: Mainlining databases: supporting fast transactional workloads on universal columnar data file formats. Proc. VLDB Endow. **14**(4), 534–546 (2020)
31. Li, T., Chandramouli, B., Burckhardt, S., Madden, S.: DARQ matter binds everything: performant and composable cloud programming via resilient steps. Proc. ACM Manag. Data **1**, 1–27 (2023)
32. Li, T., Chandramouli, B., Faleiro, J.M., Madden, S., Kossmann, D.: Asynchronous prefix recoverability for fast distributed stores. In: Proceedings of the 2021 International Conference on Management of Data, SIGMOD'21, New York, NY, USA, pp. 1090–1102. Association for Computing Machinery (2021)
33. Liu, R., Zhang, H., Chen, H.: Scalable read-mostly synchronization using passive reader–writer locks. In: USENIX Annual Technical Conference (2014)
34. Lomet, D.B.: Process structuring, synchronization, and recovery using atomic actions. In: Proceedings of an ACM Conference on Language Design for Reliable Software, New York, NY, USA, pp. 128–137. Association for Computing Machinery (1997)
35. Lu, Y., Yu, X., Cao, L., Madden, S.: Epoch-based commit and replication in distributed OLTP databases. Proc. VLDB Endow. **14**(5), 743–756 (2021)
36. Makreshanski, D., Levandoski, J., Stutsman, R.: To lock, swap, or elide: on the interplay of hardware transactional memory and lock-free indexing. Proc. VLDB Endow. **8**(11), 1298–1309 (2015)
37. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys'12, New York, NY, USA, pp. 183–196. Association for Computing Machinery (2012)
38. Massalin, Henry, Pu, Calton: A lock-free multiprocessor OS kernel. ACM SIGOPS Oper. Syst. Rev. **26**(2), 8 (1992). https://doi.org/10.1145/142111.993246
39. Mellor-Crummey, J.: Concurrent queues: practical fetch-and-phi algorithms. IEEE Trans. Reliabil. **31**, 11 (1987)
40. Mellor-Crummey, J.M., Scott, M.L.: Scalable reader-writer synchronization for shared-memory multiprocessors. In: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP'91, New York, NY, USA, pp. 106–113. Association for Computing Machinery (1991)
41. Moir, M.: Transparent support for wait-free transactions. In: Mavronicolas, M., Tsigas, P. (eds.) Distrib. Algorithms, pp. 305–319. Springer, Berlin (1997)
42. Pizlo, F., Frampton, D., Hosking, A.L.: Fine-grained adaptive biased locking. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ'11, New York, NY, USA, pp. 171–181. Association for Computing Machinery (2011)
43. Prasaad, G., Chandramouli, B., Kossmann, D.: Concurrent prefix recovery: performing CPR on a database. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD'19, New York, NY, USA, pp. 687–704. ACM (2019)
44. Rajwar, R., Goodman, J.R.: Speculative lock elision: enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 34, USA, pp. 294–305. IEEE Computer Society (2001)
45. Russell, K., Detlefs, D.: Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'06, New York, NY, USA, pp. 263–272. Association for Computing Machinery (2006)
46. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC'95, New York, NY, USA, pp. 204–213. Association for Computing Machinery (1995)
47. Shirako, J., Vrvilo, N., Mercer, E.G., Sarkar, V.: Design, verification and applications of a new read-write lock algorithm. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'12, New York, NY, USA, pp. 48–57. Association for Computing Machinery (2012)
48. Thomson, A., Diamond, T., Weng S.-C., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD'12, New York, NY, USA, pp. 1–12. Association for Computing Machinery (2012)
49. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: 24th ACM Symposium on Operating Systems Principles (2013)
50. Valois, J.D.: Implementing lock-free queues. In: Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, pp. 64–69 (1994)
51. Vasudevan, N., Namjoshi, K.S., Edwards, S.A.: Simple and fast biased locks. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT'10, New York, NY, USA, pp. 65–74. Association for Computing Machinery (2010)
52. Wang, Z., Pavlo, A., Lim, H., Leis, V., Zhang, H., Kaminsky, M., Andersen, D.G.: Building a BW-tree takes more than just buzz words. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD'18, New York, NY, USA, pp. 473–488. Association for Computing Machinery (2018)
53. Wing, J.M., Gong, C.: Testing and verifying concurrent objects. J. Parallel Distrib. Comput. **17**(1–2), 164–182 (1993)
54. Zhang, L., Butrovich, M., Li, T., Nannapanei, Y., Pavlo, A., Rollinson, J., Zhang, H., Balakumar, A., Biales, D., Dong, Z., Eppinger, E., Gonzàlez, J., Lim, W.S., Liu, J., Menon, P., Mukherjee, S., Nayak, T., Ngom, A., Niu, J., Patra, D., Raj, P.G., Wang, S., Wang, W., Yu, Y.-T., Zhang, W.: Everything is a transaction: unifying logical concurrency control and physical data structure maintenance in database management systems. In: CIDR (2021)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.