Performant Almost-Latch-Free Data Structures Using Epoch Protection

Performant Almost-Latch-Free Data Structures Using Epoch Protection

Tianyu Li
litianyu@csail.mit.edu
MIT CSAIL
Cambridge, MA, USA

Badrish Chandramouli
badrishc@microsoft.com
Microsoft Research
Redmond, WA, USA

Samuel Madden
madden@csail.mit.edu
MIT CSAIL
Cambridge, MA, USA

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 modify in the face of evolving requirements. 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 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 [2, 3].

ACM Reference Format:
Tianyu Li, Badrish Chandramouli, and Samuel Madden. 2022. Performant Almost-Latch-Free Data Structures Using Epoch Protection. In *Data Management on New Hardware (DaMoN'22), June 13, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3533737.3535091

1 INTRODUCTION

Modern processors have seen a steady increase in core counts over the past several decades. Consequently, modern applications use many more threads and depend on safe and performant concurrent access to shared data structures. Traditional abstractions such as mutexes or reader-writer latches scale poorly under this environment [6, 7]. To alleviate these bottlenecks, modern systems use a wide range of techniques for latch-free programming. In particular, in recent years the idea of *epochs* [9] has become a popular way to scale highly concurrent data structures such as indexes [10, 14, 18], key-value stores [4, 15], or transaction processing systems [13, 16, 17, 19]. However, it is difficult to program and reason about epoch-based latch-free systems, as such programs typically do not have critical sections, and developers must reason about numerous interleavings to establish invariants and ensure correctness. In our own experience building FASTER [4] and Silo [17], this requires months of careful design and many lines of subtle code, followed by lengthy debugging sessions.

To make matters more complicated, there are workloads that latch-free programming cannot easily support. 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 with double the current size when growth is required. Unfortunately, this is very difficult to implement with latch-free programming; current hardware atomic instructions can only support limit-size updates (e.g, aligned 64-bit length variables), and cannot prevent interleavings of the growth operation and concurrent read/write operations. We call this type of data structure "almost-latch-free", as rare complications – in the case of the resizable array, `Push` calls that trigger resizing – prevent efficient latch-free implementation of these data structures. Programmers often need to settle for a significantly less performant alternative with a latch (e.g., protect normal list 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]). There are many more such examples in data management. For example, imagine a transactional system that compacts and reorganizes storage blocks as they become cold, but must exclude concurrent transactional access during reorganization [11], or a sharded key-value store, where threads on a shard share access to a mostly static mapping of key ownership, except when ownership transfers occur [8, 12].

In this paper, we present a general solution to the almost-latch-free challenge, called Epoch Protected Version Scheme (EPVS). EPVS introduces lightweight critical sections programmers can use to protect vulnerable operations of the almost-latch-free data structure while adding minimal overhead to the other scalable concurrent operations. At the heart of EPVS is FASTER's highly optimized epoch protection framework that can sustain tens of millions of fine-grained operations per second. The simplest way

DaMoN'22, June 13, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9378-2/22/06.
https://doi.org/10.1145/3533737.3535091

to use EPVS is as a more scalable replacement for the standard read-write latch. For advanced users, EPVS allows complex orchestration of concurrent operations and long-running critical sections through a general state machine model. For example, [15] and [11] 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 AND RELATED WORK

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.

Epochs were first introduced by Kung and Lehman [9] as an efficient garbage collection mechanism for concurrent binary search trees. Garbage collection remains an important use case of epochs in more modern concurrent data structures such as the Bw-tree [10, 14, 18], but epochs have been used in a more general sense as well. Silo [17] 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 [13]. For this paper, we adopt a more general notion of epoch protection for thread coordination and safe code scheduling, as formulated in FASTER [4], which we will introduce in detail.

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 and 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 Figure 1. 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 Figure 1. Consider a thread t executing from line 6 while a separate renewal thread starts executing from 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.

```
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());
```

Figure 1: Example Use Case of Epoch Protection Framework

3 EPOCH-PROTECTED VERSION SCHEMES

While the previously mentioned epoch protection framework is promising for many use cases, it is insufficient for the almost-latch-free problem. The key missing ingredient here 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 may cause some concurrent updates to be missing from the new array. This is a key limitation of epochs as formulated in [4], and one we address with our proposed framework of EPVS.

3.1 Version Schemes

We introduce the idea of *versions* to augment epochs. Protected regions of code execute from start to finish within a certain version, similar to epochs. However, unlike epochs, versions have defined *transitions* associated with each version change. For an almost-latch-free data structure like the resizable list, it operates in "stable state" in most cases, and allows for latch-free concurrent operations. When operations such as resizing happen, the data structure undergoes a version change, which changes the data structure and limits concurrent operations, until it is in stable state again.

The simplest way to perform a version change is to execute a transition function in a critical section where no concurrent protected code is allowed to execute. Intuitively, we execute concurrent operations such as reads and writes in protected regions, and resize the array by bumping the epoch. The epoch framework already ensures that the resize will not start until concurrent operations have finished; to additionally ensure mutual exclusion, we enforce that future operations do not start until the resize is finished, which we accomplish through a boolean flag that threads spin-wait for when resizing is underway. Note that 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 Section 6. This tradeoff, however, results in better overall performance as long as version changes are infrequent.

3.2 Transition State Machines

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., resizing an already large array), all concurrent work is paused and scalability will suffer. This issue is further exacerbated if completion of the critical section is dependent on some external signal (e.g., completion of a background task, persistence of data on disk). The source of this issue is that we assumed it is unsafe to execute protected regions outside of the

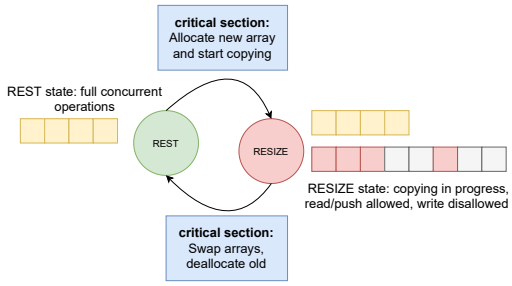


Figure 2: Resizable array with two-phase state machine – adding an intermediate `RESIZE` state allows for concurrent read and push operations

stable state. Consider again the example of resizable arrays; it is possible to define an intermediate, semi-stable state of `RESIZE` and allow limited forms of operations while the array is resizing. We summarize the logic in Figure 2, where `REST` is the stable state as before. In `RESIZE`, 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 index range $[0, c)$, as writers must be aware of the copying progress to decide whether they should write to `arr_old` or `arr_new`, which may result in excessive thread synchronization on the copying progress; any such writers must wait until `REST`. Then, because no writes are allowed to happen, any reads into index range $[0, c)$ can proceed as normal to `arr_old`. 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 or written 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 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 the 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 flag that indicates 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. 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, our simple version scheme shown in the last subsection 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.

3.3 Putting it Together: The EPVS API

To summarize, EPVS builds on top of a general epoch protection framework to support critical sections and transition state machines. This allows programmers to largely retain the performance of latch-free programming on the common, highly concurrent operations in almost-latch-free data structures, while being able to easily handle the occasional expensive calls with critical sections. We summarize the external facing API of EPVS below:

- `CurrentState()`: Peek at current state
- `Enter()`: Begin executing protected region; protected region will execute entirely within the returned state
- `Leave()`: Exit protected region
- `AdvanceVersion(VersionSchemeStateMachine)`: Execute the given state machine to advance the version (asynchronously). Only one state machine can be active at a given time.

Detailed example code using EPVS is available in Appendix A.

4 EXAMPLES

We now briefly sketch how EPVS can be used to solve many almost latch-free problems in data management to showcase its generality.

Cluster Ownership Management: Modern key-value stores like FASTER [4] are often highly scalable on a single node. However, most practical workloads require that a key-value store be distributed across multiple shards; this results in the need to enforce shard key ownership and ensure safety during ownership transfers. In short, shards will need to validate their ownership of a key before performing operations. In the common case, ownership is static and validation is a simple lightweight step. However, 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. With EPVS, we can use a simple one-state version scheme to update the ownership mapping and execute each request entirely within a protected region. This ensures safety during ownership transfers without impacting thread scalability on a single node.

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 [11], 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. [11] solves this problem with a 4-stage transformation process integrated with the DBMS's concurrency control system [19]. With EPVS, the same algorithm can be rewritten as a EPVS state machine. When the system decides that a storage block should be transformed, it registers a version change with EPVS, with an intermediate state where transformation is underway, and transactional writes are temporarily blocked.

Asynchronous Checkpointing: Traditional DBMS recovery methods using a write-ahead log lead to a serial bottleneck for update-intensive workloads. To alleviate such bottlenecks, one can utilize the *concurrent prefix recovery* (CPR) model to asynchronously and incrementally checkpoint DBMS state, and each CPR checkpoint can be regarded as executing a EPVS state machine. Again using the FASTER system as an example — in the common code path, FASTER threads modify entries in-place; however, during checkpointing, the database is temporarily frozen while its contents are flushed to storage, and updates go through a special read-copy-update path. Using EPVS, threads enter WAIT_FLUSH state and determine the checkpoint content (i.e., the portion of the database to freeze) in the transition critical section. Threads in WAIT_FLUSH read-copy-update, and transition the system back to rest when flush is complete. Note that while this algorithm is similar to the one originally presented in [15], it is noticeably simpler with only two states (three if we were to follow the strict model in [15], where threads also have to wait for I/Os issued prior to the checkpoint to complete first). This is because the original algorithm was built directly on top of the epoch framework, and must guard against more interleavings due to the lack of critical sections.

5 IMPLEMENTATION

We first describe FASTER's efficient, latch-free implementation[1] of the epoch protection framework sketched in Section 2 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 can locally cache the array offset for their entry without a hash table lookup. We additionally 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, and 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.

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, 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.

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:

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

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

We use a single 64-bit atomic `State` struct to store a version number, current state, and a flag on whether a thread intends to execute a critical section (intermediate state flag). `Enter` call acquires epoch protection and checks the current state, and suspends and retries until the state is not intermediate. To start a state machine, the `AdvanceVersion` call tries to atomically swap in a `VersionSchemeStateMachine` object, and retries later if a state machine is already underway. 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) checking 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

The primary challenge in implementing EPVS is ensuring that the system 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, each time we step the state machine, we include an additional step attempt. For state machines that may delay a step (e.g., until copying is done in resizable array), we require that the state machine itself triggers the step when a step becomes available.

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 [5], using the machine type Standard_D48s_v3, which has 48 vCPUs (roughly equivalent to 2.30 GHz Intel Xeon E5-2673 v4 CPUs) across 2 CPU sockets.

6.1 End-to-End Experiments

Cluster Ownership Management: Figure 3 shows a comparison of our prototype of a sharded version of the FASTER key-value store, which first validates incoming requests for ownership before executing. We track ownership similar to Redis-cluster using 16-bit hash buckets; ownership information for each shard is stored in a local C# `ConcurrentDictionary` object. 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.

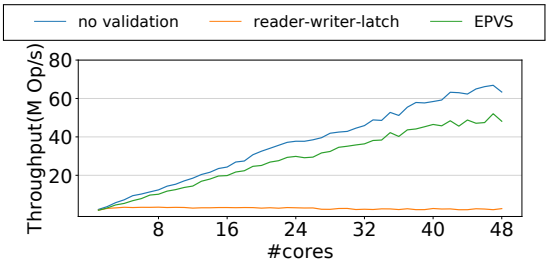
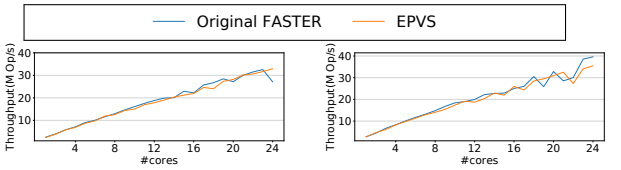


Figure 3: Scalability of FASTER-remote key validation

Asynchronous Checkpointing: We now compare EPVS against a hand-written latch-free epoch-based checkpointing solution as described in [15]. 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 16384 operations. As shown in Figure 4, 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 critical section is underway. We can see this manifest in the coarse-grained operations case in Figure 4, 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 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.

Resizable Array: Lastly, Figure 5 compares throughput of 4 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 Section 3, as simple-EPVS and 2-phase-EPVS,



(a) 500ms checkpoints, fine-grained epoch

(b) 500ms checkpoints, coarse-grained epoch

(c) 25ms checkpoints, fine-grained epoch

(d) 25ms checkpoints, coarse-grained epoch

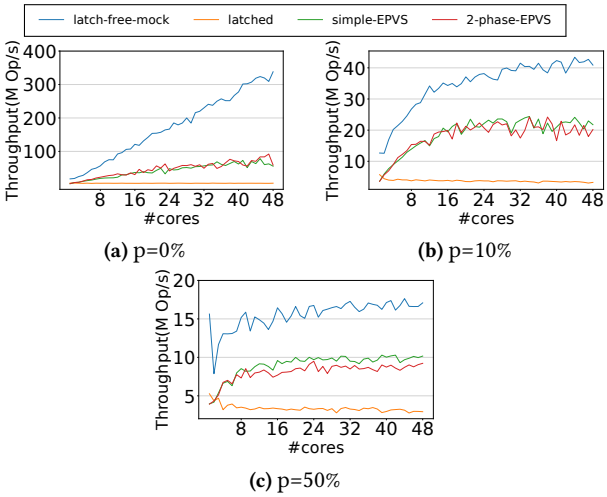
Figure 4: Scalability of FASTER-KV checkpointing

respectively. We randomly generate one million operations with $p\%$ push operations, and $(1-p)/2\%$ each for read and write operations. The array starts at 1 million elements and we issue 1 million operations per thread. We can see in Figure 5 that EPVS-based solutions are as scalable as the latch-free baseline, and perform significantly better than the latched alternatives. That said, overall throughput using EPVS is lower than the latch-free-mock due to the intrinsic overhead of epochs (a few random memory access), which is expensive compared to the extremely fast array operations (one random memory access). As we show in other experiments, this cost is negligble for many other workloads. For pushes, it is also worth noting that we would expect the latch-free-mock to perform twice as well as other solutions, due to the added amortized cost of resizing. Additionally, push itself cannot be a linearly scalable operation as push percentage increases, as there is workload-induced contention at array tail. Although the two-phase-EPVS array implementation is in theory superior to the simple-EPVS one, resizing is exponentially rare (and still relatively fast) that it does not affect overall latency or throughput in any appreciable way. It is also important to note that 2-phase-EPVS displays no advantage over simple-EPVS. This is due to increased overhead in the state machine logic (comparatively expensive as the protected region is a simple memory access) and the relatively infrequent and cheap resizing. 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.

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)



(a) p=0%

(b) p=10%

(c) p=50%

Figure 5: Scalability of EPVS Resizable Array

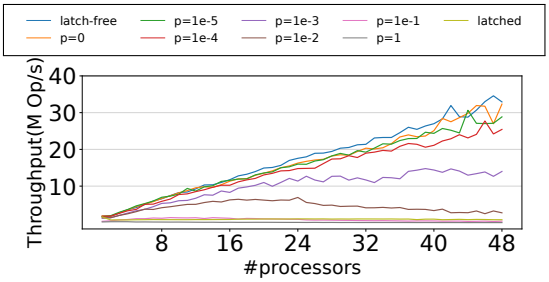
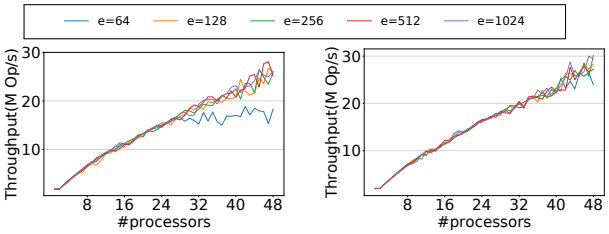


Figure 6: Sensitivity Analysis of Version Change Frequency

or with mutual exclusion (latched). We can see from the results in Figure 6 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. 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.

Impact of Epoch Table Size: Recall from Section 5 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 Figure 7 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



(a) enter-leave **(b)** refresh

Figure 7: Sensitivity Analysis of Epoch Table Size

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 Figure 7, and see that indeed this improves scalability for smaller table sizes.

7 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].

ACKNOWLEDGMENTS

We are grateful for the support of the MIT DSAIL@CSAIL member companies.

REFERENCES

[1] 2021. Intel® oneAPI Threading Building Blocks (oneTBB) Documentation for concurrent_vector. https://www.intel.com/content/www/us/en/develop/documentation/onetbb-do.
[2] 2022. Epoch Protected Version Scheme (source code). https://aka.ms/epvs.
[3] 2022. Microsoft FASTER. https://github.com/microsoft/FASTER.
[4] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM. http://doi.acm.org/10.1145/3183713.3196898
[5] Marshall Copeland, Julian Soh, Anthony Puca, Mike Manning, and David Gollob. 2015. *Microsoft Azure: Planning, Deploying, and Managing Your Data Center in the Cloud* (1st ed.). Apress, USA.
[6] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 981–992. https://doi.org/10.1145/1376616.1376713
[7] 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* (Saint Petersburg, Russia) *(EDBT '09)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/1516360.1516365
[8] Chinmay Kulkarni, Badrish Chandramouli, and Ryan Stutsman. 2021. Achieving High Throughput and Elasticity in a Larger-than-Memory Store. *Proc. VLDB Endow.* 14, 8 (apr 2021), 1427–1440. https://doi.org/10.14778/3457390.3457406
[9] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (sep 1980), 354–382. https://doi.org/10.1145/320613.320619
[10] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. 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)*. IEEE Computer Society, USA, 302–313. https://doi.org/10.1109/ICDE.2013.6544834
[11] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. 2020. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14, 4 (dec 2020), 534–546. https://doi.org/10.14778/3436905.3436913
[12] Tianyu Li, Badrish Chandramouli, Jose M. Faleiro, Samuel Madden, and Donald Kossmann. 2021. *Asynchronous Prefix Recoverability for Fast Distributed Stores*. Association for Computing Machinery, New York, NY, USA, 1090–1102. https://doi.org/10.1145/3448016.3458454
[13] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-Based Commit and Replication in Distributed OLTP Databases. *Proc. VLDB Endow.* 14, 5 (jan 2021), 743–756. https://doi.org/10.14778/3446095.3446098
[14] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855
[15] Guna Prasaad, Badrish Chandramouli, and Donald Kossmann. 2019. Concurrent Prefix Recovery: Performing CPR on a Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 687–704. https://doi.org/10.1145/3299869.3300090
[16] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838
[17] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*.
[18] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 473–488. https://doi.org/10.1145/3183713.3196895
[19] Ling Zhang, Matthew Butrovich, Tianyu Li, Yash Nannapanei, Andrew Pavlo, John Rollinson, Huanchen Zhang, Ambarish Balakumar, Daniel Biales, Ziqi Dong, Emmanuel Eppinger, Jordi Gonzàlez, Wan Shen Lim, Jianqiao Liu, Prashanth Menon, S.S. Mukherjee, Tanuj Nayak, Amadou Ngom, Jeff Niu, D. Patra, P. Govind Raj, Stephanie Wang, Wuwen Wang, Yao-Tin Yu, and William Zhang. 2021. Everything is a Transaction: Unifying Logical Concurrency Control and Physical Data Structure Maintenance in Database Management Systems. In *CIDR*.

A CODE SAMPLES

In this appendix, we showcase the C# EPVS API and several example programs written with EPVS. These are meant to be simplistic representations of EPVS and the depicted applications for illustration purposes, rather than runnable code. Complete examples are available as part of the FASTER repository [3].

A.1 EPVS

```
struct State
{
  long GetVersion() {...}
  byte GetPhase() {...}
  ...
}

interface StateMachine
{
// Given the current statem compute the next state
// this version scheme should enter, or false if no
// such transition is available yet
  bool GetNextStep(State current, out State next);

// Logic to execute before entering a state.
// Guaranteed to execute under mutual exclusion with
// other transition logic or EPVS-protected regions
  void OnEnteringState(State from, State next)

  ...
}

class EpochProtectedVersionScheme
{
// Enter protected region under the returned state
  State Enter() {...}

// Equivalent to leaving and entering, but likely
// cheaper
  State Refresh() {...}

// Leave previouslyprotected region
  void Leave() {...}

// Attempt to start executing the given state machine.
// May spin if a state machine is underway. Returns
// false if state machine has ToVersion() that is
// smaller than current version. Blocks until state
// is back to REST if block is set to true.
  bool ExecuteStateMachine(StateMachine sm,
                           bool block) {...}

  ...
}

// A simple state machine that transitions from REST
// to REST with the given critical section
class SimpleStateMachine : StateMachine {
  Action criticalSection;

  SimpleStateMachine(Action a) {...}


  override bool GetNextStep(StateMachine current,
                            out StateMachine next) {
    Debug.Assert(current.Phase == REST);
    next = new State(REST, current.Version + 1);
    return true;
  }

  override void OnEnteringState(State from,
                                State to) {
    Debug.Assert(from.Phase == REST
              && to.Phase == REST);
    criticalSection();
  }
}
```

Figure 8: Summary of EPVS API

A.2 Ownership Transfer Protection

```

class KeyValueStore {
// Handle some request for key k
  Response Process(Key k, Request r) {...}

  ...
}

class KeyValidator {
// Not safe to invoke concurrently with ownership
// updates
  bool OwnsKey(Key k);

  void UpdateOwnership(...);
}

class KeyValueServer {
  ...
  KeyValueStore kv;
  KeyValidator validator;
  EpochProtectedVersionScheme epvs;
  ...

// for normal key-value operations
  void HandleNormalRequest(Key k, Request r) {
    Response response;
    epvs.Enter();
    try {
      if (!validator.OwnsKey(k))
// error handling
        ...
      response = kv.Process(k, r);
    } finally {
      epvs.Leave();
    }
// Send response
    ...
  }

// For ownership transfers
  void HandleTransferRequest(...) {
    epvs.ExecuteStateMachine(new SimpleStateMachine(
        () => validator.UpdateOwnership(...)), true);
  }

}
```

Figure 9: Example implementation of safe ownership transfer using EPVS

A.3 Resizable Array

```
class ResizableArray<T> {
  EpochProtectedVersionScheme epvs;
  T[] list;
  int count;

  ...

  long Read(int index) {
    epvs.Enter();
    try {
// bounds check need to include list.length
// because push may increment count beyond
// current list size
      if (index >= 0 && index < count
          && index < list.Length)
        return list[index];
// Exception handling for out-of-bounds requests
      ...
    } finally {
      epvs.Leave();
    }
  }

  int Push(long value) {
// Atomically obtain position to push to
    var pos = Interlocked.Increment(ref count) - 1;
    epvs.Enter();
    try {
// Need to perform resize check in a loop because
// more than one resize may be required to reach pos
      while (true) {
// Normal push into array tail
        if (result < list.Length) {
          list[result] = value;
          return result;
        }
// Issue resize from the first entry to run
// out of space to avoid duplicate calls
        if (result == list.Length)
          epvs.ExecuteStateMachine(
            new SimpleStateMachine(
              () => /* Resize */ ...));
// refresh continuously if blocked to ensure progress
        epvs.Refresh();
      }
    } finally {
      epvs.Leave();
    }
  }
  ...
}
```

Figure 10: Resizable Array Implementation Using EPVS with a Simple One-Phase State Machine

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

  ListGrowthStateMachine(ResizableArray<T> obj) {...}


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

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

Figure 11: Two-Phase State Machine for Resizable Array as Depicted in Section 3

```
class ResizableArray<T>
{
  EpochProtectedVersionScheme epvs;
  T[] list, newList;
  int count;
  ...
  public long Read(int index) {
    var state = epvs.Enter();
    try {
      if (index < 0 || index >= count)
// Error handling
        ...
      if (index < list.Length)
        return list[index];
// If in COPYING phase, entry may have been
// pushed onto the new list
      if (state.Phase == COPYING
        && index < newList.Length)
        return newList[index];
// Error handling
      ...
    } finally {
      epvs.Leave();
    }
  }

  public int Push(T value) {
    var pos = Interlocked.Increment(ref count) - 1;
    var state = epvs.Enter();
    try {
      while (true) {
        if (state.Phase == REST
          && pos == list.Length) {
          epvs.ExecuteStateMachine(
            new ListGrowthStateMachine(this));
          state = epvs.Refresh();
        }
// Wait until the list to push to can accommodate
// pos of this element
        var listTarget = state.Phase == REST
          ? list
          : newList;
        if (pos >= listTarget.Length) {
            state = epvs.Refresh();
            continue;
        }
// if copying, need to copy to old list first to
// prevent copying from erasing the write.
        if (state.Phase = COPYING
          && pos < list.Length)
          list[pos] = value;
        listTarget[pos] = value;
      }
    } finally {
      epvs.Leave();
    }
  }
  ...
}
```

Figure 12: Resizable Array Implementation Using EPVS with Two-Phase State Machine

A.4 FASTER Checkpointing State Machine

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

  override bool GetNextStep(State curr, State next) {
    switch(curr.Phase) {
      case REST:
        next = new State(LOG_FLUSH, curr.Version)
        return true;
      case LOG_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 LOG_FLUSH:
// Complete checkpoint by resetting local data
// structures and persisting metadata
        ...
        break;
    }
  }
}

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

Figure 13: Refactored FASTER checkpoint logic with EPVS. The original logic is omitted due to space limitations.