Optimistic Concurrency with OPTIK

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Vasileios Trigonakis *
EPFL
vasileios.trigonakis@epfl.ch

Abstract

We introduce OPTIK, a new practical design pattern for designing and implementing fast and scalable concurrent data structures. OPTIK relies on the commonly-used technique of version numbers for detecting conflicting concurrent operations. We show how to implement the OPTIK pattern using the novel concept of OPTIK locks. These locks enable the use of version numbers for implementing very efficient optimistic concurrent data structures. Existing state-of-the-art lock-based data structures acquire the lock and then check for conflicts. In contrast, with OPTIK locks, we merge the lock acquisition with the detection of conflicting concurrency in a single atomic step, similarly to lock-free algorithms. We illustrate the power of our OPTIK pattern and its implementation by introducing four new algorithms and by optimizing four state-of-the-art algorithms for linked lists, skip lists, hash tables, and queues. Our results show that concurrent data structures built using OPTIK are more scalable than the state of the art.

1. Introduction

Building concurrent data structures (CDSs) in a *pessimistic* manner is easy, but typically does not lead to good performance. For example, one can design a linked list protected by a global lock in a few minutes, but inevitably, this list will be non-scalable. Accordingly, *optimistic concurrency* is deployed in every state-of-the-art data structure (e.g., lists [19, 29], hash tables [8, 37], trees [4, 41], queues [39, 40]). With optimistic concurrency, operations perform some non-synchronized work, before employing synchronization (i) for validating this optimistic work, and (ii) for possibly modifying the data structure. Performing non-synchronized work allows concurrent threads to execute truly in parallel.

Nevertheless, optimistic concurrency additionally introduces the need for validating the non-synchronized parts of the operation in order to detect conflicting concurrent operations. Validating this optimistic work is far from being trivial. Every new scalable CDS algorithm introduces a new neat technique for efficiently handling validation. Concrete examples are the linked list by Tim Harris [19] that marks the least significant bit of a pointer to indicate deletions, as well as the binary search tree by Natarajan et al. [41] that marks edges instead of nodes to minimize the number of stores. These techniques are great, but are very specific to the corresponding data structure and are thus hardly generalizable to other structures.

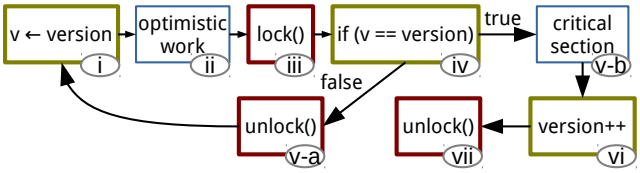

Figure 1. The OPTIK pattern (high-level view).

Ideally, general-purpose design patterns could assist developers in creating efficient CDSs. Design patterns are commonplace in software engineering as they allow for easy and efficient solutions to recurring problems. In concurrent programming, commonly-used software constructs such as locks, semaphores, monitors, and thread pools can be viewed as design patterns. However, these patterns are very low level. Higher-level patterns are required for systematically designing and implementing efficient CDSs.

Of course, there are techniques for simplifying the design of optimistic CDSs that could be viewed as high-level design patterns. The read-copy update (RCU) [35] technique stipulates wait-free reads and safe memory reclamation. RCU is intended for read-dominated workloads and might result in low scalability in the presence of updates. Software transactional memory (STM) [46] can be also used to design CDSs. Nevertheless, due to the instrumentation and the metadata overhead of STMs, the resulting CDSs are typically slower than their lock-free or lock-based counterparts. Hardware transactional memory (HTM) [26] removes the instrumentation overhead of STMs by keeping track of the transactional metadata in hardware. Although HTM already exists in commercial processors [30], it is not yet ubiquitous. Additionally, naively implementing CDSs with HTM results in non-scalable implementations (e.g., accessing the whole list within a transaction) [50].

In this paper, we introduce OPTIK,[1] a new pattern for designing and implementing fast and scalable concurrent data structures. OPTIK relies on version numbers for detecting concurrency. A version number is coupled with a lock that protects a set of data (e.g., one list node). The version number has the same granularity as the lock, thus we can devise both coarse- and fine-grained algorithms with OPTIK. An optimistic operation, such as an insertion in a hash-table bucket, uses the version number in the following steps (Figure 1): (i) it locally stores the current value of the version in order to later use it for validation, (ii) it performs some optimistic, non-synchronized work, (iii) it grabs the corresponding lock, (iv) it validates that the version number has not changed, (v-a) if validation fails, it releases the lock and restarts the operation, otherwise (v-b) it performs the critical-section work, and then (vi) it increments the version number to indicate to other threads that the protected data has been modified, and finally, (vii) it releases the lock.

* Author names appear in alphabetical order.

PPoPP '16 March 12-16, 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/2851141.2851146

[1] The name OPTIK stands for "*optimistic concurrency with ticket locks*," as our first implementation of OPTIK locks builds on top of ticket locks.

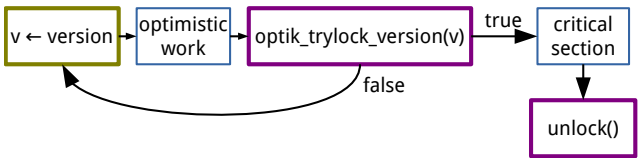

Figure 2. The OPTIK pattern implemented with OPTIK locks.

Intuitively, the validation in step (v) can fail because the version number has been incremented between steps (i) and (iii). This alteration indicates that a concurrent thread completed a modification on the protected data, rendering the optimistically accessed data inconsistent. Naturally, the reader might wonder about (a) the genuineness of OPTIK, and (b) why it has not been recognized in the past as a pattern for designing CDSs.

Version numbers have been extensively used in databases [32], STMs [6, 9], and distributed systems [1, 12]. However, as we more thoroughly discuss in the related-work section (§6), we are the first to recognize that the underlying idea can be expressed in a general way that offers a fast technique for detecting concurrency in CDSs. We argue that the main reason why the OPTIK pattern has not appeared in the past is the lack of an efficient implementation.

Consider the steps taken in Figure 1. To detect concurrency with versions, we must include the "overhead steps" (i), (iv), and (vi). To make things even worse, if validation in step (iv) fails, the thread has acquired the lock, possibly after contending for it, just to fail the validation and restart. To the best of our knowledge, most existing state-of-the-art lock-based algorithms, such as the linked-list by Heller et al. [22] and the skip list by Herlihy et al. [29], include exclusively the overhead for step (iv), namely for validating that the optimistic results are still consistent.[2] Consequently, implementing the OPTIK pattern as described above, would not only include the same overheads as existing algorithms, but also the ones for keeping track of and incrementing the version numbers.

We solve the aforementioned limitations of the OPTIK pattern by introducing the OPTIK-lock abstraction that merges locking with validation. OPTIK locks rely on the simple observation that existing lock algorithms, such as ticket locks, employ version numbers in their implementation. Accordingly, we design the OPTIK-lock abstraction that offers an extended interface to traditional locks. In particular, OPTIK locks offer the `optik_trylock_version(lock, targetv)` function that acquires the lock iff (a) the `lock` is free, and (b) the current version in the `lock` is the same as the `targetv` version. We concretely implement the OPTIK-lock abstraction on top of ticket locks.[3] As the unlock function of ticket locks simply increments the version, we can also merge unlocking with incrementing the version number.

Accordingly, as we show in this paper, we can efficiently implement the OPTIK pattern using OPTIK locks (Figure 2). The resulting implementation guarantees that if the lock is acquired, then the critical section will be performed. Therefore, we are able to reduce contention behind the lock and to avoid the wasted work of waiting for the lock only to fail the validation. Locking and validation are performed with a single compare-and-swap. In a sense, OPTIK locks bring lock-based algorithms closer to their lock-free counterparts, where validation and the actual modifications are performed in single steps with atomic operations.

We illustrate the effectiveness of OPTIK by (a) designing new algorithms and by (b) optimizing existing state-of-the-art ones for linked lists, hash tables, skip lists, and queues. In particular, we design two new linked list algorithms, one based on global and one on fine-grained locks, and we introduce the concept of *node caching* for speeding up list traversals. Based on these lists, we design two corresponding hash tables. Additionally, we design a new concurrent array map and use it in a hash table, and we employ OPTIK locks in optimizing existing hash tables. Furthermore, we use OPTIK locks to simplify validation in the optimistic skip-list algorithm [29] and we design a novel, simple skip-list algorithm based on the OPTIK pattern. Finally, we design three variants of the classic Michael-Scott queues [39] and we also introduce the concept of *victim queues* for reducing enqueue contention. Our OPTIK-lock library, together with the data structures we design and optimize with OPTIK are available at `http://go.epfl.ch/optik`.

The main contributions of this paper are as follows:

- We identify the OPTIK design pattern that can be used to easily design and optimize concurrent data structures.
- We introduce OPTIK locks that offer a concrete and efficient implementation of the OPTIK pattern.
- We design four new highly-efficient data-structure algorithms and optimize four existing state-of-the-art algorithms.

We focus in this paper on using OPTIK in CDSs. Nevertheless, we could imagine using OPTIK, instead of the classic lock interface, wherever a lock can be used. The only requirement is that the critical section must include a read-only prefix that can be optimistically performed before acquiring and validating the OPTIK lock. Of course, we do not claim that OPTIK is a silver bullet for all concurrency problems, but rather that it is an efficient design pattern for various use cases. For example, OPTIK locks are not very suitable for protecting large chunks of data that can be independently updated (e.g., the next pointers of a node of a large skip list). In these cases, OPTIK can lead to false validation failures due to updates on unrelated data (e.g., §5.3). Additionally, an OPTIK lock comprises a single memory location, thus, as every lock algorithm, it can become a scalability bottleneck if heavily stressed (e.g., §5.4).

The rest of the paper is organized as follows. In §2, we recall some background notions on CDSs. We describe the OPTIK pattern/lock and use them in two concrete examples in §3 and §4, respectively. We then illustrate how to use OPTIK in designing and optimizing various CDSs in §5. We discuss related work in §6, and we conclude the paper in §7.

2. Concurrent Data Structures

Data structures allow for efficient storage and retrieval of data elements. These elements are typically identified by unique keys. In particular, *search data structures* (e.g., lists, hash tables) include three main operations: (i) *search*, for searching for an element with a given key, (ii) *insert*, for inserting a new element in the structure if the key is not already there, and (iii) *delete*, for deleting an existing element. Other data structures, such as queues, offer a different interface. Queues are FIFO structures with two main operations: (i) *enqueue*, to place an element at the head of the queue, and (ii) *dequeue*, to remove the current tail element (if any).

Concurrent data structures (CDSs) can be simultaneously accessed by multiple threads through their interface. The consistency of CDSs is typically measured with respect to linearizability [28]. Linearizable CDS algorithms are commonly classified based on the progress guarantees they offer. It is common to distinguish between *blocking* [27], *lock-free*, and *wait-free* [25] algorithms. Blocking algorithms typically rely on *locking* and might block, waiting for a lock to be released. Lock-free algorithms are non-blocking in the sense that (i) they do not employ locks, and (ii) they ensure that at least one process in a system can make progress. Finally, wait-free algorithms are also non-blocking and guarantee that every process in a system eventually makes progress. In practice, wait-free algo-

[2] Each algorithm uses custom techniques for implementing this validation.

[3] As we show in §3, we can implement OPTIK locks on top of other lock algorithms as well.

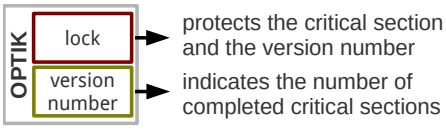

Figure 3. The basic building block of the OPTIK pattern.

rithms are slower than their lock-based and lock-free counterparts and are thus not very commonly used.

Most state-of-the-art CDS algorithms are *optimistic*, regardless if they are lock-based or lock-free. They are optimistic in the sense that they first optimistically perform some work, without synchronizing with other threads, and then synchronize to validate the consistency of the optimistic work and to modify the state of the structure. Lock-based algorithms do so using locks and validation within the critical section, while lock-free algorithms use atomic operations, such as compare-and-swap, to simultaneously validate and update the target nodes of the data structure.

3. OPTIK

In this section, we detail the OPTIK pattern, we present the OPTIK-lock abstraction, and we describe two concrete implementations of the OPTIK-lock abstraction. We also then discuss practical considerations regarding implementing and using OPTIK, such as lock nesting, memory barriers, and memory reclamation.

3.1 The OPTIK Pattern

As we point out in §1, the OPTIK pattern relies on version numbers to detect potentially conflicting concurrency (see Figure 1). As Figure 3 shows, this version number is coupled with a lock and shares the same granularity as that lock (i.e., it protects the same data). The version number is incremented upon every successful critical section that modifies the shared protected state. Thus, intuitively, we can detect whether there were concurrent modifications on the protected state if we observe a version change.

Accordingly, with OPTIK we can implement some sort of a transaction (we discuss this resemblance with transactions below), where we read the version number before starting the optimistic part of the transaction. Then, whenever we want to modify the protected data, we acquire the lock and check whether the version number is still the same. If that is the case, then no other thread could have completed a concurrent operation. Otherwise, we know that at least one thread has concurrently committed a modification.

Because the version number has the same granularity as the corresponding lock, we might have false conflicts. For example, in a linked list protected by a global lock (see §5.1), every committed modification conflicts with any concurrent one, although they might operate on completely unrelated parts of the list. In practice, in most cases we can control the granularity of the lock, hence the granularity of the version number.[4]

The OPTIK pattern has three main strengths. First, it offers a concrete way of "thinking" about optimistic concurrency, similar to STMs. With an STM, the designer makes use of transactions, but then it is up to the STM runtime to optimistically execute and coordinate these transactions. In contrast, with OPTIK, the designer must explicitly delimit the optimistic and the synchronized parts of an operation. Still, she does not need to rely on ad-hoc techniques, such as marking pointers, for validating the optimistic results. Second, the OPTIK pattern has a concrete, fast implementation based on OPTIK locks. If the pattern is appropriately employed, the resulting CDS will be efficient and scalable (as we show in §5). Third, in our experience (see §4 and §5), OPTIK-based CDSs are simpler and easier to prove correct than the state of the art. In many OPTIK-based CDSs, the linearization point of an insertion or deletion is the actual write that makes a node physically linked or unlinked.

OPTIK *vs. STM Transactions.* The OPTIK pattern can be viewed as a transaction. OPTIK shares some common characteristics with traditional STM transactions, especially those that defer synchronization to the commit phase (e.g., [6, 9, 14]). First, they are both explicitly delimited (i.e., we know where the transaction begins and where it ends). Second, they both include an optimistic phase. Finally, the optimistic phase is followed by a validation/commit phase where conflicting concurrency is typically detected. If there are conflicts, then both OPTIK and STM transactions are restarted, otherwise they commit their modifications. For instance, OPTIK transactions are very similar to the ones of NOrec STM [6]. NOrec employs a global lock that is further used as a version number for validation, in a way similar to OPTIK.

However, in contrast with STMs, OPTIK does not offer isolation or atomicity guarantees. STM transactions are typically *opaque* [18] (i.e., they are serializable and they disallow even non-committed transactions from accessing inconsistent state). OPTIK allows transactions to access the intermediate results of other ongoing transactions. Additionally, STM transactions typically provide all-or-nothing semantics (i.e., *atomicity*). With OPTIK, a transaction can partially complete and then restart. The atomicity control is fully up to the programmer. Precisely because of this lack of guarantees, OPTIK can operate with zero instrumentation overhead.

3.2 The OPTIK-Lock Abstraction

The OPTIK-lock abstraction merges locking with version-number validation in a single atomic step.[5] By doing so, we can implement the OPTIK pattern without the extravagant overhead of locking and then failing the validation (compare Figures 1 and 2). OPTIK locks extend the traditional lock interface with various functions. The most important ones are listed and explained below:

- `optik_trylock_version(lock, targetv)` [non-blocking]: acquires the `lock` iff the `lock` is free and the version of the `lock` is the same as in `targetv`. Returns a boolean indicating whether the `lock` was acquired.
- `optik_lock_version(lock, targetv)` [blocking]: acquires the `lock` and returns a boolean that shows whether the version that was acquired is the same as in `targetv`.
- `optik_unlock(lock)` [non-blocking]: increments the `lock`'s version number and releases the `lock`.
- `optik_revert(lock)` [non-blocking]: reverts the version of the `lock` to the one before acquiring the `lock`. It can be used to release the `lock` when no modifications were performed in the critical section.
- `optik_get_version(lock)` [non-blocking]: returns the current version of the `lock`.
- `optik_get_version_wait(lock)` [blocking]: waits until the `lock` is free and returns its current free version.
- `optik_is_same_version(v0, v1)` [non-blocking]: returns a boolean on whether versions `v0` and `v1` are the same.
- `optik_is_locked(v)` [non-blocking]: returns a boolean on whether version `v` is locked.

We provide two implementations of the OPTIK-lock abstraction, one on top of ticket and one on top of versioned locks. For brevity, we detail the versioned-lock-based implementation (as it is simpler than the one on top of ticket locks) and discuss the additional functionality that OPTIK locks on top of ticket locks offer. In principle, the OPTIK-lock abstraction can be implemented on top of more

[4] Skip lists are somewhat of an exception to this rule (see §5.3).

[5] Of course, it is up to the corresponding implementation of the abstraction to guarantee this single-step locking and validation.

```
1 typedef volatile uint64_t optik_t;
2 #define OPTIK_INIT      0
3 #define OPTIK_LOCKED    0x1LL //odd values -> locked

5 int optik_trylock_version(optik_t* l, optik_t targetv) {
6   if (optik_is_locked(targetv) || *l != targetv)
7     return false;
8   return CAS(l, targetv, targetv + 1) == targetv;
9 }

11 int optik_lock_version(optik_t* lock, optik_t targetv) {
12   optik_t ol_cur;
13   do {
14     do {
15       ol_cur = *lock;
16     } while (optik_is_locked(ol_cur));
17   } while (CAS(lock, ol_cur, ol_cur + 1) != ol_cur);
18   return ol_cur == targetv;
19 }

21 void optik_unlock(optik_t* lock) {
22   *lock++; // mem-release
23 }

25 void optik_revert(optik_t* lock) {
26   *lock--; // mem-release
27 }

29 int optik_is_locked(optik_t v) {
30   return (v & OPTIK_LOCKED);
31 }

33 optik_t optik_get_version(optik_t* lock) {
34   return *lock; // mem-acquire
35 }

37 optik_t optik_get_version_wait(optik_t* lock) {
38   do {
39     optik_t olv = *lock; // mem-acquire
40     if (!optik_is_locked(olv))
41       return olv;
42   } while (1);
43 }

45 int optik_is_same_version(optik_t v1, optik_t v2) {
46   return v1 == v2;
47 }
```

Figure 4. Code for OPTIK locks on top of versioned locks.

lock algorithms. Nevertheless, `optik_trylock_version` is in the heart of the OPTIK pattern, thus we argue that every OPTIK-lock implementation must provide atomic (i.e., single compare-and-swap) locking and validation. Such an implementation requires base lock algorithms which incorporate version numbers.

OPTIK Locks Using Versioned Locks

An OPTIK lock (`optik_t`) is just an 8-byte unsigned counter (`uint64_t` in C). An odd value for the counter indicates that the lock is *locked*, while an even value means *unlocked*. The acquire function tries, until successful, to compare-and-swap (CAS) the current (even) value v with $v + 1$. The release function simply increments the counter value. Figure 4 includes the concrete implementation of the OPTIK abstraction on top of versioned locks. We briefly discuss this implementation.

First, `optik_trylock_version`, the most important OPTIK function, returns false (lines 6-7) if the lock is already locked or if the current lock version is not the same as the target version (`targetv`). The former check is necessary for correctness, otherwise the operation might try to erroneously CAS an odd value to an even one. The latter check is an optimization for avoiding unnecessary CAS invocations.

Similarly, `optik_lock_version` spins while the lock is locked and the tries to acquire the lock with a CAS. The unlock and revert

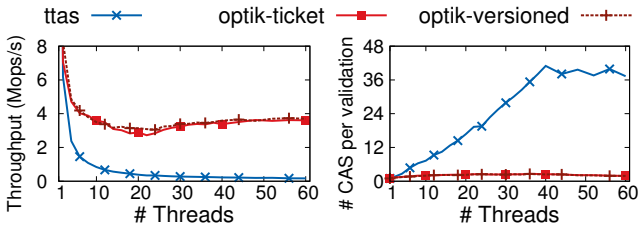

Figure 5. Locking and validation with and without OPTIK locks.

functions increment and decrement the lock value, respectively, to indicate that the lock is now free and that a modification was (not) performed. The `optik_is_locked` function simply checks whether the given version is an odd number. The `get_version` and `get_version_wait` functions return the current version of the lock. The latter spins while the lock is locked and only then returns the version number. Finally, `optik_is_same_version` compares whether two version numbers are equivalent.

OPTIK Locks Using Ticket Locks

Ticket locks have a number of very unique properties. First, although they typically occupy just 8-bytes:

```
struct ticket_t { uint32_t ticket, current; };
```

they are fair. To acquire the lock, the thread grabs a ticket `t` with an atomic fetch-and-increment and waits until `t==lock->current`. To unlock, the owner simply increments the lock's `current` field.

Additionally, ticket locks show the amount of queuing behind the lock. For instance, if `lock->ticket - lock->current` equals to three, then the lock is occupied and two more threads are queued, waiting to acquire the lock. We can use this information to take decisions depending on the levels of contention (see §5.4 for an example). Finally, each thread is able to check how far from acquiring the lock it is in the queue. We can use this information to implement efficient backoff schemes. Note that we are not the first to exploit these properties of ticket locks [7, 31, 36].

Based on these properties of ticket locks, OPTIK locks offer the `optik_num_queued` and the `optik_lock[_version]_backoff` extensions. The former returns the number of threads waiting for the lock, while the latter implements waiting with backoff that is proportional to the distance of the thread from acquiring the lock.

A shortcoming of OPTIK on top of ticket locks compared to the implementation over versioned locks is the 32-bits long version number of the former. If a thread stores the version number and then "sleeps" for 2^{32} lock acquisitions, then the version number could overflow, resulting in a potentially incorrect validation.[6] In contrast, OPTIK locks on top of versioned locks require 2^{63} acquisitions while the thread is sleeping (two increments per acquisition).

The OPTIK Pattern with and without OPTIK Locks

We illustrate the necessity of OPTIK locks with an experiment. We compare the throughput of a single OPTIK lock with the throughput of implementing version validation without OPTIK locks. As we explain earlier, to validate the version number without OPTIK locks the thread must always acquire the lock. We implement this behavior using 8 bytes; 4 bytes for a test-and-test-and-set (TTAS) lock and 4 bytes for the version number. The version number is validated and incremented while holding the lock.

Figure 5 depicts the validated lock-acquisition throughput with and without OPTIK locks, as well as the average number of CAS operations that are executed per successful validation on an Intel Xeon

[6] If the lock delivers 100M acquires/s, which is almost impossible on modern hardware [7], the thread must sleep for ∼40s for the overflow to happen.

server (see §5 for platform details and our experimental settings). The two OPTIK-lock implementations behave identically and deliver significantly higher throughput than validating with normal locking. OPTIK locks are more than 10 times faster than TTAS on average, explained by the number of CAS invocations per validation that grows significantly with TTAS due to lock contention.[7] As we explain earlier, without OPTIK locks the threads might wait behind the lock to later fail the validation.

3.3 Practical Considerations

OPTIK *with Lock Nesting.* The OPTIK pattern offers the "read then lock-validate version" functionality for a single OPTIK lock. Lock nesting (i.e., acquiring and holding more than one lock at a time) requires acquiring the locks one after the other. Therefore, although the validation of an earlier lock succeeds, the validation of a later one might fail. For example, it might happen that:

`optik_trylock_version(`l_1`, `v_1`) → true;`
`optik_trylock_version(`l_2`, `v_2`) → false.`

Depending on the semantics of the algorithm, failing the second `optik_trylock_version` can have different outcomes. For example, on the delete operation of a linked list (see §4.2 for details), failing the second trylock results in restarting the whole operation after reverting the first lock. On our novel OPTIK-based skip-list algorithm (see §5.3), we perform incremental insertions: Once the OPTIK lock for a skip-list level is acquired, the new node is linked to that level. If a subsequent trylock fails, the operation is restarted, but the locks for the already inserted levels are not reacquired.

OPTIK *and Memory Fences.* As we show in Figure 4, implementing OPTIK locks requires certain memory ordering guarantees when loading and storing on the shared word of the lock. In short, loading the version number (e.g., in `optik_get_version`) requires *acquire* semantics: No other memory access of the same thread can be reordered before this load. Similarly, storing on the memory of the lock (e.g., in `optik_revert`) requires *release* semantics: No other memory access of the same thread can be reordered after this store. Notice that on x86 architectures the implementation of these memory-ordering semantics does not require any memory fences.

OPTIK *and Memory Reclamation.* OPTIK decouples concurrency control from memory reclamation. Accordingly, OPTIK can be used with practically any memory-reclamation scheme, such as hazard pointers [38], RCU [35], quiescent states [19, 20]. Our CSDS implementations with OPTIK use *ssmem*,[8] a simple memory allocator with quiescent-based memory reclamation.

4. Concrete OPTIK Examples

We illustrate in detail how to use the OPTIK pattern on two examples: (i) a map structure (abstract data type), and (ii) a novel concurrent linked-list algorithm.

4.1 OPTIK-based Array Map

A map contains key-value pairs and exports the three main operations of search data structures, namely *search*, *insert*, and *delete* (see §2). We implement the map as a fixed-sized array, hence, insertions that do not find an empty spot return false (we do not employ array resizing for simplicity). In §5.2, we use our map design in a concurrent hash table.

We first briefly describe a lock-based array map that protects every operation with a global lock and then show how to optimize this array map using the OPTIK pattern.

Lock-based Map. The design of a pessimistic, lock-based array map is straightforward: All three operations grab the lock and then traverse the array. If search or delete operations find the target key while traversing, they complete the operation (i.e., read the value of the key-value pair and, for deletions only, delete the key), unlock the lock, and return. If insertions find the key while traversing, they release the lock and return false. If they do not, they insert the new key-value pair in a free spot (if any), release the lock, and return true. If no spot is empty, insertions return false.

OPTIK-*based Map.* We use the OPTIK pattern/lock to introduce optimism in the pessimistic lock-based map. Intuitively, search operations, as well as updates that return false, do not modify the data structure. Therefore, ideally, they must complete without locking. Of course, the actual insertions or deletions in the map have to synchronize for correctness.

The OPTIK pattern splits an operation into three main phases: (i) optimistic, non-synchronized (read-only) work, (ii) validation and locking, and (iii) pessimistic, synchronized (write-mostly) work. We transform the map operations to follow the three phases. Figure 6 contains the code for the concurrent OPTIK-based array map. In what follows, we describe the code step by step.

Delete. The delete operation (Figure 6(a)) follows the three phases of OPTIK. It first stores the current OPTIK version number (line 9) and traverses the array without synchronization (lines 10-19), looking for the target key (line 11). If the key is not found in the array, it just returns `NULL` without ever locking (line 20). If the key is found in line 11, then the operation tries to acquire the lock using `optik_trylock_version` with the version that was earlier stored. If the validation is successful, it deletes the key, releases the lock, and returns the value (lines 14-17). If `optik_trylock_version` fails, the operation is restarted (lines 12-13).

Insert. Insertions (Figure 6(b)) follow very similar logic with deletions. If the key is found while traversing the array, the operation returns false without ever acquiring the lock. If not, it tries to acquire the lock with `optik_trylock_version` and, if successful, it performs the insertion (if there is a free array spot).

Search. We want the search operation to be lock-free, otherwise, the total throughput of the map will be dictated by the maximum lock throughput. Nevertheless, we must guarantee the atomicity of reading key-value pairs. In other words, we have to ensure that

```
1 typedef struct {
2   key_t key;
3   val_t val;
4 } key_val_t;
5
```

```
typedef struct {
  key_val_t* array;
  size_t size;
  optik_t* lock;
} map_t;
```

```
7 val_t optik_map_delete(map_t* map, key_t key) {
8  restart:
9   optik_t vn = optik_get_version(map->lock);
10  for (int i = 0; i < map->size; i++) {
11    if (map->array[i].key == key) {
12      if (!optik_trylock_version(map->lock, vn))
13        goto restart;
14      map->array[i].key = NULL;
15      val_t val = map->array[i].val;
16      optik_unlock(map->lock);
17      return val;
18    }
19  }
20  return NULL;
21 }
```

(a) Delete operation of OPTIK-based concurrent map.

[7] If we use a test-and-set lock instead of a TTAS, the number of CAS per validation "explodes."

[8] *ssmem* is available at `https://github.com/LPD-EPFL/ssmem`.

```
1 int optik_map_insert(map_t* map, key, val) {
2  restart:
3   optik_t vn = optik_get_version(map->lock);
4   int free_idx = -1;
5   for (int i = 0; i < map->size; i++) {
6     key_t curr_key = map->array[i].key;
7     if (curr_key == key) { return false; }
8     else if (curr_key == 0) { free_idx = i; }
9   }

11  if (!optik_trylock_version(map->lock, vn))
12    goto restart;

14  int res = false;
15  if (free_idx >= 0) {
16    map->array[free_idx].key = key;
17    map->array[free_idx].val = val;
18    res = true;
19  }
20  optik_unlock(map->lock);
21  return res;
22 }
```

(b) Insert operation of OPTIK-based concurrent map.

```
1 val_t optik_map_search(map_t* map, key_t key) {
2  restart:
3   optik_t vn = optik_get_version_wait(map->lock);
4   for (int i = 0; i < map->size; i++) {
5     if (map->array[i].key == key) {
6       val_t val = map->array[i].val;
7       optik_t vnc = optik_get_version(map->lock);
8       if (optik_same_version(vn, vnc))
9         return val;
10      goto restart;
11    }
12  }
13  return NULL;
14 }
```

(c) Search operation of OPTIK-based concurrent map.

Figure 6. An OPTIK-based concurrent array map data structure.

between matching an array key with the target key and reading the value, there was no concurrent modification on this key-value pair.

We achieve this guarantee using the OPTIK version number. The search operation (Figure 6(c)) reads the version number in the beginning of the operation (line 3), like update operations do. This time, however, we employ the `optik_get_version_wait` function that blocks until the lock is free. Once the key is matched (line 5), we read the corresponding value and check whether the version has changed (lines 6-8). If it did change, then the operation is restarted, otherwise the value is returned. The reason for acquiring an unlocked version in line 3 is that we need to ensure that the search operation was not concurrent with any update operations on the same key during the execution of lines 5-6.

We could decrease the "granularity" of the version number for search operations, by reading the version before line 5. We would still be able to acquire atomic snapshots of the key-value pairs. However, doing so puts a lot of stress on the cache line of the OPTIK lock, resulting in lower performance than the design in Figure 6. (Actually, we can devise various schemes for validating the key-value snapshot using the version number.)

In terms of correctness, successful updates are serialized behind the lock. Successful search operations are trivially correct, as they complete iff there were no concurrent modifications. Unsuccessful search operations can be linearized so that they never observe the target key. Similarly, unsuccessful updates can be linearized so that they do (or do not) observe the target key.

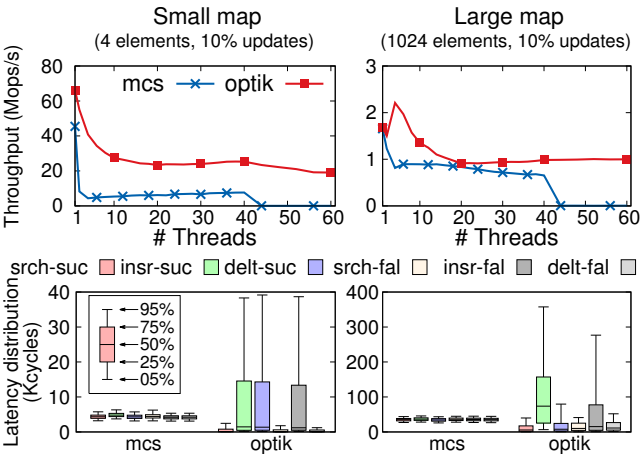

Figure 7. Lock-based vs. OPTIK-based map. The latency-distribution results are taken on 10 threads.

Lock-based vs.* OPTIK*-based Array Map. We compare the two map implementations on two workloads on an Intel Xeon (see §5 for platform details and our experimental settings). Figure 7 depicts the results, where *mcs* represents the lock-based map protected by an MCS lock [36]. On both the small and the large maps, the OPTIK version (*optik*) is faster than the lock-based one. *optik* has two main benefits compared to *mcs*. First, search operations (80% of the workload) do not acquire the lock. Second, unsuccessful updates (∼10% of the operations) also do not need to synchronize.

If we exclude the results on multiprogramming (i.e., more threads than hardware contexts), where *mcs* suffers, *optik* is on average 4.7 and 1.4 times faster than *mcs* on the small and the large map, respectively. On the small workload, since there are just four spots in the map array, many operations fail (e.g., deletions do not find the key they are looking for). For example, on 10 threads, only 25% of the updates are successful, resulting in 5% total effective updates. Overall, the results can be largely explained by the latency distributions. *optik* significantly reduces the latencies for search operations and for unsuccessful deletions. The reduction is less profound on unsuccessful insertions, as a portion of those failures is due to insufficient space in the array. In these cases, both *optik* and *mcs* acquire and release the lock before returning false. Additionally, the effects of failing `optik_trylock_version` and restarting are visible on the tail latencies of successful updates.

4.2 OPTIK-based Linked List

The main idea behind a sorted OPTIK-based linked list is to keep track of the necessary version numbers while traversing the list. In a sense, similarly to hand-over-hand locking (also known as lock coupling) [27], the OPTIK-based list performs hand-over-hand version tracking. Figure 8 includes the code of our implementation. We defer the evaluation of our list to §5.

Delete. The delete operation (Figure 8(a)) is the most complex operation of the OPTIK-based linked list, because it requires locking two nodes; the one being deleted and its predecessor node. Traversing the list (lines 11-15) keeps track of these two version numbers that are later used for locking with `optik_trylock_version` (lines 18-23). If locking the predecessor node fails, the operation is restarted, otherwise the node to be deleted is locked. If this latter `optik_trylock_version` fails, the predecessor's OPTIK lock is reverted, instead of unlocked, in order to avoid false conflicts with other concurrent operations. Notice that due to OPTIK, (i) no

deleted flag is required (as in [22]), and (ii) the OPTIK lock of the deleted node is never released, which prohibits updates from reusing this node. Essentially, the linearization point of a deletion is the actual write on the `pred->next` pointer in line 24.

Insert. Inserting in the OPTIK-based linked list (Figure 8(b)) requires locking and validating only the predecessor node (line 12). This OPTIK lock ensures that there are no concurrently completed modifications on the predecessor node `p` or on `p->next`.

Search. The search operation (Figure 8(c)) of the OPTIK-based linked list is completely oblivious to concurrency. We can support this 100% sequential search design because the linearization points of updates are the actual stores on the predecessor node.

5. OPTIK in Concurrent Data Structures

In this section, we illustrate the power and usefulness of OPTIK for optimizing and designing concurrent data structures (CDSs) (i.e., linked lists, hash tables, skip lists, and queues). In contrast to §4, we keep the CDS descriptions high-level for brevity. Before that, we describe the evaluation settings that we use in our experiments and the two platforms that we evaluate our data structures on.

Experimental Methodology. We evaluate various algorithms via microbenchmarks. Unless stated otherwise, all OPTIK implementations use OPTIK locks on top of versioned locks. Similarly, unless stated otherwise, non-OPTIK implementations use test-and-set locks. (Notice that for highly-contented locks, such as the locks in concurrent queues, we use MCS locks.) We take the non-OPTIK algorithm implementations from the ASCYLIB library [8] (we use the optimized versions of the algorithms). Additionally, we use the memory allocator of ASCYLIB that provides garbage collection and we use 8-byte long keys and values. Backoff schemes can significantly affect the performance of CDSs (e.g., when an operation fails and must be restarted). For fairness, all data structures use

```
1 typedef struct node {
2   key_t key; val_t val;
3   optik_t lock;
4   struct node* next;
5 } node_t;

7 val_t optik_ll_delete(ll_t* list, key_t key) {
8  restart:
9   node_t *pred, *cur = list->head;
10  optik_t predv = curv =
        optik_get_version(&cur->lock);
11  do {
12    pred = cur; predv = curv;
13    cur = cur->next;
14    curv = optik_get_version(&cur->lock);
15  } while (cur->key < key);
16  if (cur->key != key) { return NULL; }

18  if (!optik_trylock_version(&pred->lock, predv))
19    goto restart;
20  if (!optik_trylock_version(&cur->lock, curv)) {
21    optik_revert(&pred->lock);
22    goto restart;
23  }
24  pred->next = cur->next;
25  val_t result = cur->val;
26  optik_unlock(&pred->lock);

28  node_gc_free(cur);
29  return result;
30 }
```

(a) Delete operation of OPTIK-based concurrent linked list.

```
typedef struct {
  node_t* head;
} ll_t;
```

```
1 int optik_ll_insert(ll_t* list, key, val) {
2  restart:
3   node_t *pred, *cur = list->head;
4   optik_t predv = curv = OPTIK_INIT;
5   do {
6     curv = optik_get_version(&cur->lock);
7     pred = cur; predv = curv;
8     cur = cur->next;
9   } while (cur->key < key);
10  if (cur->key == key) { return false; }

12  if (!optik_trylock_version(&pred->lock, predv))
13    goto restart;
14  node_t* newnode = new_node(key, val, cur);
15  pred->next = newnode;
16  optik_unlock(&pred->lock);
17  return true;
18 }
```

(b) Insert operation of OPTIK-based concurrent linked list.

```
1 val_t optik_ll_search(ll_t* list, key_t key) {
2   node_t* cur = list->head;
3   while (cur->key < key) { cur = cur->next; }
4   if (cur->key == key)  { return cur->val; }
5   return NULL;
6 }
```

(c) Search operation of OPTIK-based concurrent linked list.

Figure 8. An OPTIK-based linked-list data structure.

the exact same backoff function. We use exponentially increasing backoff times with up to 16k cycles maximum backoff. Furthermore, after every iteration, threads wait for a short duration, in order to avoid long runs [39].

On every run, we set the initial size of the data structure and the key range that the threads operate on. On every iteration, each thread selects a key at random within the given range. We keep the range double the initial size and the percentages of insertions and deletions the same, so that the size of the structure remains close to the initial. Because the key range is double the initial, roughly half of the update operations on search data structures return false. The update rate that we report on the graphs represents the effective percentage of updates, namely the ones that alter the data structure. For our skewed workloads, we use a zipfian distribution of keys with $a = 0.9$, where the largest keys are the most popular. Our results are the median value of 11 repetitions of 5 seconds each. We do not pin threads to cores, but let the OS do the scheduling.

For our latency measurements, we use the the per-core timestamp counter [30] for accurately measuring the duration of an operation in cycles. In detail, every thread holds an array of 16K latency measurements that, in the end of each experiment, are collected and translated to latency distribution (boxplots reporting 5^{th}, 25^{th}, 50^{th}, 75^{th}, and 95^{th} percentile latencies).

We use the following two multi-cores:

Xeon. The 20-core Intel Xeon consists of two sockets of Xeon E5-2680 v2 Ivy-Bridge 10-core (20 hyper-threads). It runs at 2.8 GHz and includes 32 KB, 256 KB, and 25 MB (per die) L1, L2, and LLC, respectively.

Opteron. The 48-core AMD Opteron contains four Opteron 6172 multi-chip modules (MCMs). Each MCM has two 6-core dies, for a total of 8 memory nodes. It operates at 2.1 GHz and has 64 KB, 512 KB, and 5 MB (per die) L1, L2, and LLC data caches.

5.1 OPTIK in Linked Lists

We use OPTIK in the design of concurrent (sorted) linked lists. The simplest algorithm is of course a sequential list protected by a

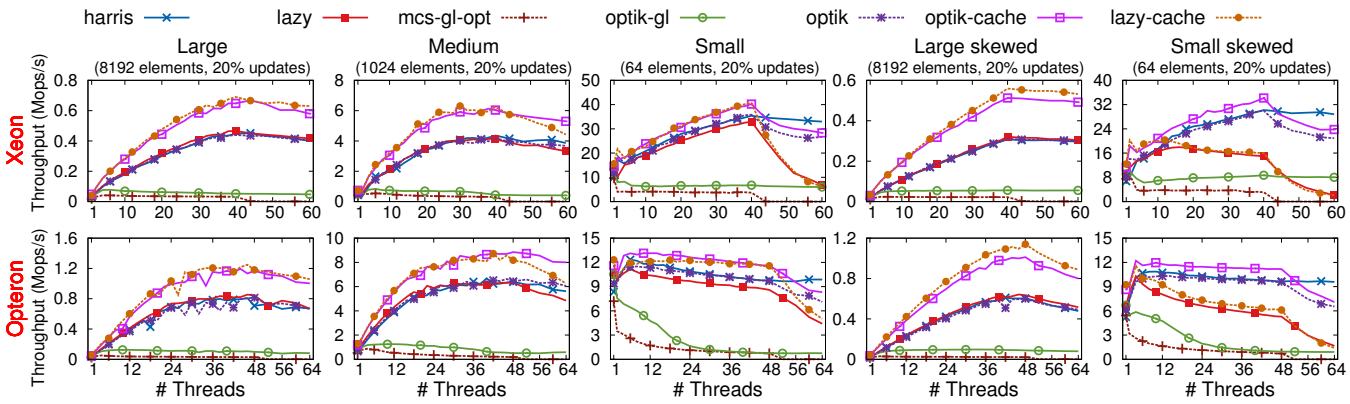

Figure 9. Throughput of linked-list algorithms on Xeon and Opteron on various workloads.

scalable global lock, such as an MCS lock. Naturally, this algorithm does not offer any concurrency as all operations are serialized behind the lock. An easy optimization on the global-lock algorithm is to implement the search operation so that it does not acquire the lock (given that memory reclamation is properly handled). The linearization point of updates is then the actual memory writes that access the predecessor node of the one being updated.

Nevertheless, updates are fully serialized behind the global lock, resulting in low scalability. We can alleviate this with OPTIK by introducing optimism to the update operations. The transformation is very similar to that of the concurrent map in §4.1. Note that concurrent modifications might not be conflicting, still, using a global lock will result in false conflicts. Because of this limitation and of the high load on the global lock, this linked-list design is not expected to scale well on contended scenarios. We can resolve these limitations using fine-grained locking (see §4.2 for the design of the fine-grained OPTIK-based linked list).

Additionally, inspired by the fact that version numbers reveal whether a list node has been modified, we develop the idea of *node caching*. In short, each thread keeps track of the last accessed node after each operation, accompanied by the version number that the thread observed. This node can be subsequently used as the entry point for the next operation on the list, given that (i) it has not been deleted, and (ii) it is a correct entry point (i.e., in a sorted list, the key of the cached node is less than the target key). Of course, we must ensure that the memory of deleted nodes is not re-used while the node is still referenced by any node cache. Node-caching can be also applied on non-OPTIK algorithms, given that we can avoid the ABA problem and that we can detect whether a node is valid.

Correctness. The OPTIK-based global-lock list is trivially correct as it disallows concurrency of modifications. The linearization point of both insertions and deletions can be set to the actual write on the predecessor's next pointer. Search operations either observe the concurrent modifications in the vicinity of the target key, or not.

Evaluation. Figure 9 depicts the throughput of the aforementioned linked-list algorithms on various workloads. For comparison, we include the results of the lazy linked-list algorithm [22] (*lazy*), that has been shown to be very efficient [8, 16], as well as the lock-free list by Harris [19] (*harris*). We implement the node-caching idea on the lazy list (*lazy-cache*) and on the fine-grained OPTIK-based list (*optik* and *optik-cache* in the graph). *mcs-gl-opt* represents a global-lock list protected by an MCS lock, including the non-synchronized search optimization we describe earlier.

Clearly, the node-cache optimization (*optik-cache, lazy-cache*) brings important performance benefits as it probabilistically reduces the list-traversal duration. For instance, on the large list, 49.8% of the operations make use of the node cache, while on the small list the hit rate drops to approximately 40%. On these two workloads, *optik-cache* delivers 50% and 15% higher average throughput than the version without the cache (*optik*).

Additionally, the OPTIK-based global-lock list (*optik-gl*) delivers higher throughput than *mcs-gl-opt* in all workloads. *optik-gl* mostly benefits from the fact that for 20% of the operations–the unsuccessful ones–it returns without acquiring the lock.

Finally, the fine-grained OPTIK-based list (*optik*) performs similarly to *lazy* and *harris* for the low-contention workloads (i.e., large, large-skewed, and medium). However, *optik* is more scalable than lazy on high-contention levels. On 64 elements, *optik* is on average 22% faster than *lazy*. Note that *optik* stresses the locks less than *lazy*, because the operations do not acquire the lock if they are going to fail the validation. This difference is clear on the small-skewed workload, where neither *lazy*, nor *lazy-cache* can sustain the contention of the highly-contented nodes.[9] Additionally, *optik* behaves much better than *lazy* on multiprogramming and is, on average, just 5% slower than *harris* even on the small workloads.

5.2 OPTIK in Hash Tables

We adapt and use the two OPTIK-based linked lists (§4.2, §5.1) in the design of two novel hash tables. Intuitively, the list protected by a global lock, resulting in per-bucket locking, is more suitable for hash tables. We also use the array map of §4.1 in the design of a third hash table.

We further use OPTIK locks to optimize existing hash tables. In a hash table, an update operation (i.e., an insertion or a deletion) might not be feasible: Delete (resp. insert) operations return false if the corresponding key is not found (resp. is found). Many hash-table algorithms (e.g., Java `ConcurrentHashMap` [34]) implement updates by directly locking the corresponding bucket, regardless if the operation is feasible. This unnecessary locking hinders scalability [8]. In these algorithms, in order to return false without locking if an update is not feasible, we must add an extra read-only traversal of the bucket. If the operation cannot be performed, no lock is acquired and the operation simply returns false after this first traversal. Otherwise, if the operation can be performed, we must acquire the bucket lock and then re-traverse the bucket to ensure that no concurrent modification operated on the target key. Consequently, for every successful update, we have two traversals of the bucket. We can avoid the second traversal with OPTIK locks, using either `optik_lock_version` or `optik_trylock_version`. In the beginning of the operation, we keep track of the version number of the bucket and use this version in the OPTIK-lock call. If the version is

[9] The most contented node is accessed by 15% of the requests.

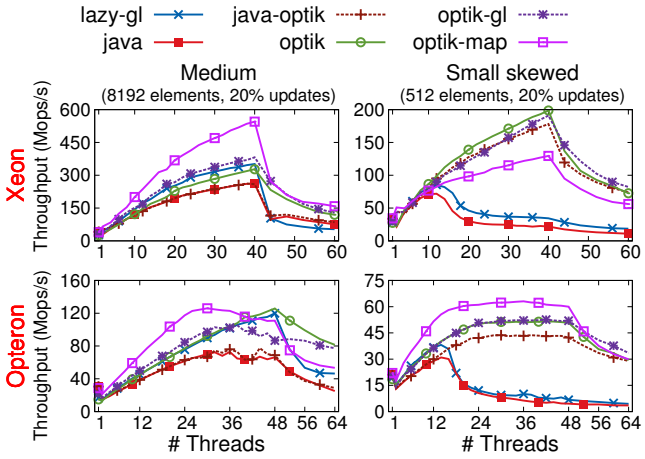

Figure 10. Throughput of hash-table algorithms on Xeon and Opteron on various workloads.

validated, no concurrent modification has completed on this bucket, hence we do not need to re-traverse the bucket.

Correctness. The three hash tables that are based on the two OPTIK lists and the map are correct because of the correctness of these base data structures. The optimizations for avoiding double traversal with OPTIK are correct because the bucket cannot be modified without increasing the version number of the bucket lock.

Evaluation. Figure 10 includes the results of various hash tables. We set the number of buckets to be equal to the number of initial elements, so that initially every bucket contains on average one element. In the interest of space, we only show the results with medium and small-skewed sized hash tables. On the missing graphs, the behavior of the hash tables is in accordance with the results shown in Figure 10. Apart from the three OPTIK-based hash tables (*optik*, *optik-gl*–for per-bucket locking, and *optik-map*), we create a hash table with lazy linked lists adapted to use per-bucket locking (*lazy-gl*). Additionally, we evaluate Java's `util.concurrent.ConcurrentHashMap` [34] (*java*), as well as a modified version that avoids double parsing using `optik_trylock_version`, as we describe above. The `ConcurrentHashMap` algorithm uses lock striping: It partitions the buckets into n segments. Each segment (and its buckets) is protected by a single lock and can be individually resized. We configure n to be 128, based on Java's documentation [42] "*Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table.*"

Optimizing *java* with OPTIK (*java-optik*) brings benefits only in the presence of (high) contention. On the large hash table (65536 elements–not shown in the graph), the improvement is just 1.9%, because there are practically no validation failures. Additionally, the second pass on the bucket of *java* is very fast, as the first pass brings the bucket data in the L1 cache of the core.

Furthermore, *optik-map* does not scale well on the small workloads on Xeon due to the hardware. In brief, the buckets of *optik-map* are allocated in consecutive memory locations, thus occupying a few contiguous cache lines, resulting in increased hardware prefetching on Xeon in our experiments. For example, on 20 threads, the small hash table triggers three orders of magnitude more last-level-cache prefetches than the medium one. This inaccurate prefetching leads to low scalability due to high coherence traffic. Once the size of the hash table is large enough, *optik-map* becomes the fastest hash table on both platforms. The other hash tables do not face the aforementioned problem, because they dynamically allocate each node that is inserted in the hash table.

Regarding the remaining three hash tables, *optik-gl* is the fastest. *optik-gl* is 2-times faster than *lazy-gl* on average (31% faster on the non-skewed workloads). *optik* is on average 9% slower than *optik-gl*, as for some operations *optik* acquires two locks instead of the one lock in *optik-gl*. On the small-skewed workload, we see the power of the OPTIK pattern compared to normal locking: *optik-gl* and *optik* are both 3.7-times faster than *lazy-gl* on average. Even on the large-skewed workload (not in the graph), *lazy-gl* is on average more than 2-times slower than the OPTIK-based hash tables.

5.3 OPTIK in Skip Lists

In theory, OPTIK is not very suitable for skip lists. With per-node lock granularity, the same version protects all the next pointers of the node. Consequently, validating the node with OPTIK results in false conflicts. Still, using OPTIK in skip lists results in simpler designs than the existing state-of-the-art ones [15, 29]. We first simplify validation in the optimistic skip list by Herlihy et al. [29], using `optik_lock_version`. If the validation is successful, then the corresponding node has not been modified, thus we do not need to validate the optimistic results in another way. This specific skip lists checks that the node is not logically deleted and that the next pointer at the corresponding level has not been altered.

We also use OPTIK in the design of a new skip-list algorithm. As in any skip list, update operations parse the list and keep track of the predecessor and successor nodes at each level. Due to OPTIK, parsing also keeps track of the version number of each predecessor node. These version numbers are later used for validation. Once the parsing finds the spot to modify, it locks and validates the predecessor nodes and then performs the modifications. If the validation fails, the locks are released and the operation is restarted. We implement two variants of the OPTIK-based skip list. The first one, in case validation fails, performs more fine-grained validation (same one as in [29]). The second one immediately restarts the operation if an OPTIK validation fails.

Correctness. The modified Herlihy skip list maintains the correctness of the initial algorithm. Our modifications only involve reducing validation in case the `optik_lock_version` function is able to validate the previously observed version.

For brevity, we only describe the correctness sketch of the OPTIK-based skip list that immediately restarts on a trylock failure. Both insertions and deletions traverse the list and keep track of the predecessor nodes and their version at each level. As the OPTIK lock protects the whole predecessor node `p`, we do not need to keep track of the successor nodes for validating `p->next`. Insertions try to acquire the lock and perform the insertion of the new node eagerly (i.e., they do the physical linking of the node immediately after acquiring the lock of that level). If an `optik_trylock_version` call fails, the operation is restarted and, after re-parsing the list, the insertion continues from the level that failed. A flag, similar with the *fullylinked* flag in Herlihy skip list, ensures that a partially inserted node will not be concurrently deleted. Similarly, a deletion atomically sets the flag of the target node to *deleted* and unlinks the node after acquiring all predecessor locks. We can devise a variant of the algorithm where deletions proceed progressively like insertions. However, the coordination overhead between insertions and deletions on the same node surpasses the benefits of being eager.

Evaluation. Figure 11 compares the Herlihy skip list (*herlihy*), and the lock-free one by Fraser [15] (*fraser*), with the three lists that we describe above. In the interest of space, we only show the results on large-skewed and small-skewed lists. On low-contention levels (large, medium non-skewed–not shown in the graph), all algorithms behave similarly. Intuitively, all five implementations follow almost identical code paths in the absence of conflicts: Most of the time is spent traversing the list.

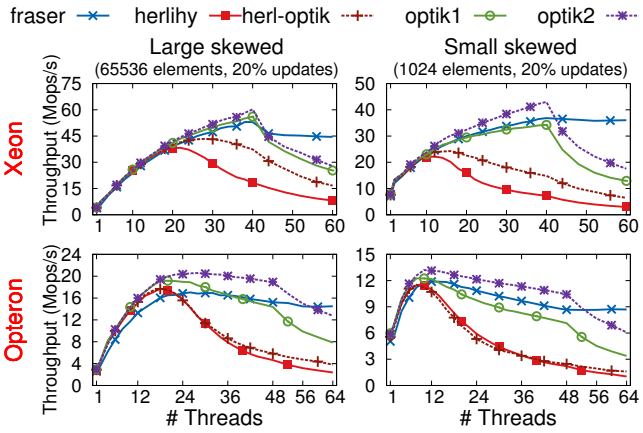

Figure 11. Throughput of skip-list algorithms on Xeon and Opteron on various workloads.

Using `optik_lock_version` in the Herlihy skip list (*herl-optik*) slightly affects the performance on the Opteron, but has a large effect on Xeon. In brief, the faster validation with OPTIK results in an important reduction of operation restarts. For instance, on the small-skewed workloads, on 20 threads on Xeon, without OPTIK 30% of update operations have to restart due to concurrency, compared to 24% with OPTIK. Contrarily, on the Opteron due to the overall lower throughput than Xeon, both *herlihy* and *herl-optik* have 50% operation restarts on 20 threads.

On skewed workloads, we also notice the benefits of using OPTIK, even though it can introduce unnecessary operation restarts. In particular, *optik2*, which is the variant that immediately restarts if there is a trylock failure, is more scalable than *optik1*, that uses `optik_lock_version` and does fine-grained validation if the version is not validated. For example, on very-high contention, on 20 threads, 40% of the operations have to restart with *optik2*, while just 20% with *fraser*. Still, *optik2* delivers 10% higher throughput than *fraser* on 20 threads. The main reason for *optik2* being more scalable than the rest is the important property of OPTIK that we have already extensively discussed: Threads fail the validation with a single atomic operation, without waiting behind the occupied lock. The other three lock-based skip lists do not include false restarts, they do however include false contention behind the per-node locks. *optik2* also benefits from (i) simpler implementation than the rest, as it does not include the fine-grained validations, and (ii) the eager node insertion. Overall, *optik2* is faster than *fraser*. However, *optik2*'s throughput significantly drops on multiprogramming, while *fraser* is able to sustain its throughput.

5.4 OPTIK in Queues

We use OPTIK in various concurrent queue designs. First, we optimize the classic Michael-Scott queues [39] (MS-queue) using OPTIK locks. The first lock-based MS-queue variant employs the `optik_lock_version` function to optimize the `dequeue` function: The operation is optimistically prepared so that if the validation succeeds, only a single store is performed in the critical section. If the validation fails, the `dequeue` operation is prepared and performed in the critical section, as usual. The second (lock-based) variant is very similar to the first one, however, it uses `optik_trylock_version` instead of the lock function. If the validation fails, then the operation is restarted. The third variant is a lock-based/lock-free MS-queue hybrid. We use the lock-free `enqueue` implementation of the MS-queue unaltered. We opt for this approach because the enqueue operations do not offer any opportunities for optimism. For the `dequeue` function we use the OPTIK trylock implementation.

The final variant of MS-queue introduces the idea of *victim queues*. The `dequeue` function uses the same trylock implementation as the last two designs. The `enqueue` implementation utilizes the `optik_num_queued` function of OPTIK locks (on top of ticket locks–see §3). If the number of waiting nodes is large (e.g., more than two in our implementation), then the thread performs the insertion in a secondary victim queue, instead of waiting behind the lock. The first thread to put a node in the empty victim queue is responsible for linking the victim queue to the main one. The results are (i) lower contention behind the lock, and (ii) a simple victim-queue design as it does not interact with `dequeue` operations.

Correctness. The first three variants of MS-queue do not essentially affect the correctness of the original designs. The fourth design employs the victim-queue idea. Enqueue operations either wait behind the lock to normally perform their operation, or insert the element in the secondary victim queue. This secondary queue is linked to the main one, once the first thread to use it gets the lock. This same thread is also responsible for emptying the victim queue so it can be reused. Operations that utilize the victim queue have to wait until the victim queue has been emptied, thus their elements are visible in the main queue. This waiting ensures that they can be linearized properly.

Evaluation. We evaluate the lock-based (*ms-lb*) and the lock-free (*ms-lf*) MS-queues. We use *ms-lb* with MCS locks. We also evaluate the three MS-queue variants (*optik0*, *optik1*, *optik2*), as well as the one using victim queues (*optik3*). We initialize the queues with 65k elements. The results include several interesting points (Figure 12).

First, *ms-lb* delivers stable performance, regardless of the contention levels, due to the MCS locks. If we use any simple spinlock algorithm (e.g., test-and-set) instead of MCS, the throughput of *ms-lb* degrades as we increase contention. However, when the number of threads becomes more than the number of hardware contexts, the combination of locking and the fairness of MCS kills throughput.[10]

Second, the remaining queue algorithms do not scale and do not even keep stable performance as we increase contention, especially on Opteron. Unlike *ms-lb* with MCS locks, all other designs have two single points–cache lines–of contention, namely the head and the tail of the queue. Opteron is an 8-socket machine, thus increasing the number of threads, increases the non-uniformity as well, resulting in more expensive cache-coherence traffic [7]. Still, on both platforms, *ms-lb* is slower than the rest on less than 6-7 threads.

Third, it is worth comparing the two MS-queues with the different OPTIK-based queue implementations. *optik2* (lock-free enqueue, OPTIK-based dequeue) behaves practically the same as *ms-lf*, showing that the simple CAS validation of OPTIK locks does resemble lock-freedom. Then, the victim-queue technique of *optik3* does bring some benefits that are mostly visible on the increasing-size workload which stresses enqueues. *optik3* is on average 28% faster than *ms-lf* on this workload, while overall it is 7% faster.

Regarding *optik1*, on the one hand it contains the enqueue implementation of *ms-lb*, thus on the increasing-size workload it behaves similar to *ms-lb*. On the other hand, it uses the `optik_trylock_version` implementation for dequeuing, showing similar performance to *optik2* and *ms-lf*. Furthermore, *optik0* on the Opteron shows that using OPTIK locks with the lock/unlock interface, under high contention, is not a good idea. At the end of the day, OPTIK locks are simple spinlocks.

Finally, the latency-distribution graphs reveal the power and the weaknesses of each implementation. For example, dequeuing

[10] There are techniques, such as time-published queue-based locks [21], for alleviating this problem.

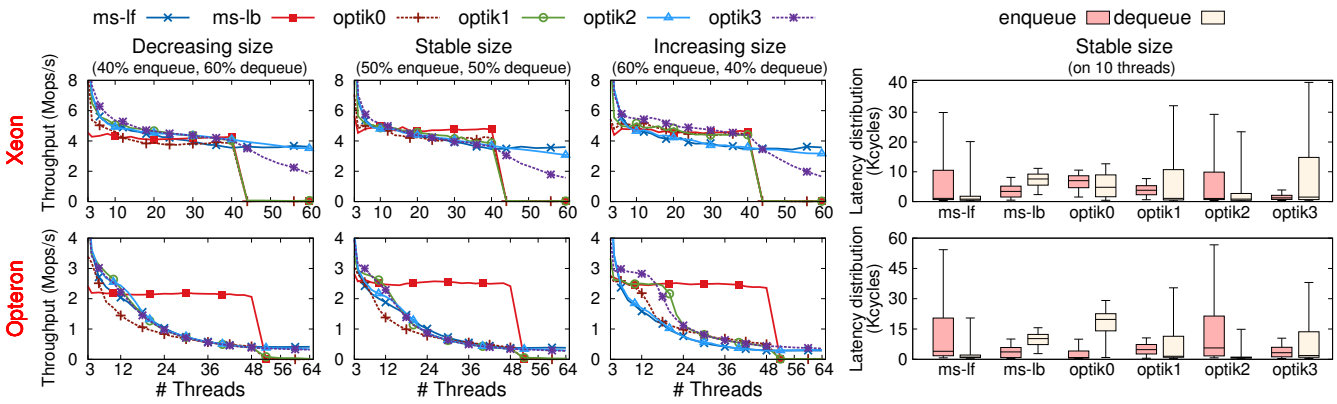

Figure 12. Throughput and latency distribution of queue algorithms on Xeon and Opteron on various workloads.

an element is very fast with *ms-lf*, however, enqueuing is very expensive. Similarly, enqueuing with *optik3* is fast because of the victim-queue approach, but dequeuing is slow.

5.5 Summary

The combination of the OPTIK pattern with OPTIK locks is a very strong concurrency tool. We illustrate the power of OPTIK by:

- designing four new CDSs: (i) an array map with a corresponding hash table, (ii-iii) a global-lock and a fine-grained linked list with two corresponding hash tables, and (iv) a skip list;
- optimizing four state-of-the-art CDSs: (i) the `ConcurrentHashMap` algorithm in Java [34], (ii) the optimistic skip list by Herlihy et al. [29], and (iii-iv) both the lock-free and lock-based Michael-Scott queues [39];
- introducing two concurrency techniques: (i) node caching for list structures, and (ii) victim queues for concurrent queues.

Of course, OPTIK is not always a suitable solution. The most prominent example of such a case is stack data structures. We briefly experiment with stacks (not shown in the graphs). More precisely, we redesign the classic lock-free stack by Treiber [48] using OPTIK. The original and the OPTIK-based variants behave similarly. Still, the contention levels that can be induced on a highly parallel stack cannot be sustained by neither the "simple" OPTIK lock, nor the lock-free solution. There are ways to alleviate this problem, such as aggressive backoff mechanisms, or elimination [24]. Note that large backoff times might result in large tail latencies.

6. Related Work

Variants of the OPTIK pattern can be basically found wherever optimistic concurrency is used (e.g., databases, distributed systems). In the following, we highlight the pieces of work that are the most-related to OPTIK.

Concurrent Data Structures (CDSs). There has been a large amount of work on designing efficient and scalable CDSs, for linked lists [19, 22, 37, 43], hash tables [8, 34, 37], skip lists [15, 43, 47], binary search trees [4, 8, 10, 13, 41], queues [39, 40, 49], and stacks [24, 48]. Every new CDS design typically introduces a new technique for detecting and handling concurrency. OPTIK is a generic design pattern that provides a way of detecting conflicting concurrency via version numbers in different CDSs.

ASCY [8] is a set of high-level principles that describe how to design scalable search data structures. Unlike ASCY, OPTIK offers a concrete design pattern and implementation. In our experience, search data structures with OPTIK do follow ASCY. In particular, the BST-TK binary search tree, part of the ASCY work, detects concurrency with version numbers (as OPTIK does). Similarly, Gramoli et al. [17] utilize version numbers to design a concurrent linked list which reduces synchronization over the lazy linked list [22]. In this paper, we generalize the usage of version numbers to a design pattern and show how to use it in various CDSs.

Optimistic Concurrency. Several concepts and tools have been proposed for designing and implementing optimistic concurrency.

Read-copy update (RCU) [35] is a technique that was introduced in the Linux kernel for easily designing CDSs with (i) wait-free reads and (ii) memory reclamation. Nevertheless, RCU targets read-mostly workloads. Arbel and Attiya [2] extend RCU to better support concurrent updates. Still, their binary-search-tree design is slower than other state-of-the-art trees, especially on write-intensive workloads. Predicate RCU (PRCU) [3] reduces the granularity of waiting in RCU. PRCU offers a tradeoff between the amount of work that search operations must do and the amount of waiting in updates. With OPTIK, we decouple memory reclamation from concurrency control, thus we are able to achieve designs that incur none of the aforementioned overheads of RCU/PRCU.

Transactional memory offers the concept of transactions for implementing synchronization. Software transactional memory (STM) [46] implements transactions in software. STM can be used in the design of CDSs, but due to the instrumentation overheads of STMs, the resulting implementations are typically slower than their lock-free or lock-based counterparts [5]. Hardware transactional memory (HTM) [26] implements transactions in hardware and thus avoids the instrumentation and the metadata overhead of STMs. Unfortunately, HTMs are currently neither ubiquitous nor robust enough to be extensively used by CDS designers.

Speculative lock elision [44, 45] aims at reducing the overhead of locking when critical sections do not actually conflict. A thread might elide a lock, meaning that threads optimistically execute their critical sections without acquiring that lock. If a true data conflict appears, then the thread rollbacks and executes the critical section normally. The main goal of lock elision is to enable writing concurrent applications with coarse-grained locking that perform well. In contrast, OPTIK's main goal is to enable the design of high-performance CDSs in a methodical way.

Flat combining [23] is another technique that appears promising for optimizing coarse-grained lock-based CDSs (e.g., queues). With flat combining, an operation translates to a message to a dedicated server thread that performs the operation on a locally-held, sequential data structure. Unlike OPTIK, flat combining is not suitable for highly-concurrent data structures, such as hash tables.

Sequence locks (seqlocks) [33] resemble OPTIK locks as they include a lock and a version number. With seqlocks, readers ensure that they read consistent data by double checking the version num-

ber. However, unlike OPTIK, seqlocks assume distinct readers/writers and keep the lock and the version separately. In fact, OPTIK locks can be used in implementing the seqlock functionality.

Version Numbers in Concurrency. Optimistic concurrency control was introduced for optimizing database transactions [32] in 1981. It relied on transaction numbers for detecting conflicting concurrency. In concurrent programming, many STM systems (e.g, TL2 [9], TinySTM [14], NOrec [6], SpecTM [11]) rely on version numbers for validating the optimistic results of transactions. Version numbers have also been employed in distributed transactions (e.g., [1, 12]) for detecting conflicts. To the best of our knowledge, we are the first to extend the traditional lock interface, with OPTIK locks, so that we merge validation with locking.

7. Conclusions

In this paper, we introduce the OPTIK design pattern and the underlying OPTIK locks. The OPTIK pattern offers a concrete and simple way of detecting conflicting concurrency in concurrent data structures. Therefore, it can be used to methodically design fast and scalable data structures. OPTIK locks provide a concrete and efficient implementation of the pattern. We illustrate the power of OPTIK by (a) designing four novel concurrent-data-structure algorithms, and by (b) optimizing four existing state-of-the-art ones.

Acknowledgments. We wish to thank our shepherd, Haibo Chen, and the anonymous reviewers for their fruitful comments. We also want to thank Tim Harris and the members of the Systems and Networking group at MSR Cambridge for their useful feedback during the early steps of OPTIK. This work has been supported in part by the European ERC Grant 339539–AOC.

References

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. SOSP '07.

[2] M. Arbel and H. Attiya. Concurrent Updates with RCU: Search Tree as an Example. PODC '14.

[3] M. Arbel and A. Morrison. Predicate RCU: An RCU for Scalable Concurrent Updates. PPoPP '15.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A Practical Concurrent Binary Search Tree. PPoPP '10.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue '08*.

[6] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. PPoPP '10.

[7] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. SOSP '13.

[8] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS '15.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. DISC '06.

[10] D. Drachsler, M. Vechev, and E. Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. PPoPP '14.

[11] A. Dragojević and T. Harris. STM in the Small: Trading Generality for Performance in Software Transactional Memory. EuroSys '12.

[12] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. NSDI '14.

[13] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking Binary Search Trees. PODC '10.

[14] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-based Software Transactional Memory. PPoPP '08.

[15] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[16] V. Gramoli. More than You Ever Wanted to Know about Synchronization. PPoPP '15.

[17] V. Gramoli, P. Kuznetsov, S. Ravi, and D. Shang. Brief Announcement: A Concurrency-Optimal List-Based Set. DISC '15.

[18] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. PPoPP '08.

[19] T. Harris. A Pragmatic Implementation of Non-blocking Linked Lists. DISC '01.

[20] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of Memory Reclamation for Lockless Synchronization. *JPDC '07*.

[21] B. He, W. N. Scherer, and M. L. Scott. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. HiPC '05.

[22] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. OPODIS '05.

[23] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. SPAA '10.

[24] D. Hendler, N. Shavit, and L. Yerushalmi. A Scalable Lock-free Stack Algorithm. SPAA '04.

[25] M. Herlihy. Wait-Free Synchronization. *TOPLAS '91*.

[26] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. ISCA '93.

[27] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised First Edition*. 2012.

[28] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *TOPLAS '90*.

[29] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A Simple Optimistic Skiplist Algorithm. SIROCCO '07.

[30] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C. 2015.

[31] S. Kashyap, C. Min, and T. Kim. Scalability in the Clouds!: A Myth or Reality? APSys '15.

[32] H.-T. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. *TODS '81*.

[33] C. Lameter. Effective Synchronization on Linux/NUMA Systems. Gelato Federation Meeting '05.

[34] D. Lea. Overview of Package util.concurrent Release 1.3.4. `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`, 2003.

[35] P. E. McKenney and J. D. Slingwine. Read-copy Update: Using Execution History to Solve Concurrency Problems. PDCS '98.

[36] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *TOCS '91*.

[37] M. M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. SPAA '02.

[38] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-free Objects. *PDS '04*.

[39] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. PODC '96.

[40] A. Morrison and Y. Afek. Fast Concurrent Queues for x86 Processors. PPoPP '13.

[41] A. Natarajan and N. Mittal. Fast Concurrent Lock-free Binary Search Trees. PPoPP '14.

[42] Oracle. ConcurrentHashMap in Java Docs. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html`, 2015.

[43] W. Pugh. Concurrent Maintenance of Skip Lists. Technical report, 1990.

[44] R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. MICRO '01.

[45] A. Roy, S. Hand, and T. Harris. A Runtime System for Software Lock Elision. EuroSys '09.

[46] N. Shavit and D. Touitou. Software Transactional Memory. PODC '97.

[47] H. Sundell and P. Tsigas. Fast and Lock-free Concurrent Priority Queues for Multi-thread Systems. *JPDC '05*.

[48] R. Treiber. Systems Programming: Coping with Parallelism. Technical report, 1986.

[49] P. Tsigas and Y. Zhang. A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. SPAA '01.

[50] L. Xiang and M. L. Scott. Software Partitioning of Hardware Transactions. PPoPP '15.