UNIVERSITÄT SALZBURG

Scal: Non-Linearizable Computing Breaks the Scalability Barrier

Christoph M. Kirsch   Hannes Payer   Harald Röck

Technical Report 2010-07   November 2010

Department of Computer Sciences

Jakob-Haringer-Straße 2
5020 Salzburg
Austria
`www.cosy.sbg.ac.at`

Technical Report Series

Scal☠: Non-Linearizable Computing Breaks the Scalability Barrier[⋆]

Christoph M. Kirsch   Hannes Payer   HaraldRöck

Department of Computer Sciences
University of Salzburg, Austria
`firstname.lastname@cs.uni-salzburg.at`

Abstract. We propose a relaxed version of linearizability and a set of load balancing algorithms for trading off adherence to concurrent data structure semantics and scalability. We consider data structures that store elements in a given order such as stacks and queues. Intuitively, a concurrent stack, for example, is linearizable if the effect of push and pop operations on the stack always occurs instantaneously. A linearizable stack guarantees that pop operations return the youngest stack elements first, i.e., the elements in the reverse order in which the operations that pushed them onto the stack took effect. Linearizability allows to reorder concurrent (but not sequential) operations arbitrarily. We relax linearizability to k-linearizability with $k > 0$ to also allow sequences of up to $k - 1$ sequential operations to be reordered arbitrarily and thus execute concurrently. With a k-linearizable stack, for example, a pop operation may not return the youngest but the k-th youngest element on the stack. It turns out that k-linearizability may be tolerated by concurrent applications such as process schedulers and web servers that already use it implicitly. Moreover, k-linearizability does provide positive scalability in some cases because more operations may be executed concurrently but may still be too restrictive under high contention. We therefore propose a set of load balancing algorithms, which significantly improve scalability by approximating k-linearizability probabilistically. We introduce Scal, an open-source framework for implementing k-linearizable approximations of concurrent data structures, and show in multiple benchmarks that Scal provides positive scalability for concurrent data structures that typically do not scale under high contention.

1 Introduction

Making data structures concurrent typically involves some form of synchronization that relies on lock-based, lock-free, or even wait-free mechanisms. The challenge has been to guarantee correctness, that is, linearizability [11], while still providing scalability. This is a highly non-trivial problem, in particular, as the number of available cores in modern systems rapidly increases. The scalability of applications is limited by Amdahl's Law, which states that the degree to which we can speed up an application on a multi-core system is limited by the amount of code that cannot be parallelized and must be executed sequentially. Since operations on shared data structures may not be

⋆ This work is supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design and the Austrian Science Fund No. P18913-N15.

fully parallelized there is an intrinsic concurrency bottleneck in many applications using shared data structures that gets increasingly problematic with an increasing number of cores.

Even basic linearizable data structures such as stacks and queues have negative scalability under high contention due to synchronization. However, it turns out that linearizability is often not needed. Consider, for example, a webserver which stores incoming requests in a shared FIFO queue running on a server machine with possibly hundreds of cores. Requests are dequeued and handled by worker threads at a later point in time. In such a scenario it is often not important to process the requests in perfect FIFO order. Instead, it may be sufficient if the order in which the requests are handled is FIFO up to a constant that bounds the deviation from FIFO order for fairness.

We therefore propose to relax linearizability to k-linearizability for trading off adherence to data structure semantics and scalability. Linearizability requires that each data structure operation takes effect at some time instant between its invocation and its response. In particular, concurrent operations, which overlap in time, may take effect in arbitrary order whereas sequential, i.e., non-overlapping operations must take effect in sequential order. The notion of k-linearizability with $k > 0$ also allows sequences of up to $k-1$ sequential operations to be reordered arbitrarily and thus execute concurrently.

We describe and analyse an open-source framework called Scal for implementing k-linearizable approximations of concurrent data structures. Instead of a single instance of a data structure Scal maintains k so-called partial data structures, which are identical instances of the original data structure, and a load balancing select function that distributes data structure operations among the k partial data structures. For example, a k-linearizable FIFO queue provides fairness up to k, i.e., it guarantees that a dequeue operation on the FIFO queue returns one of the k oldest elements. However, k-linearizability may still be too restrictive under high contention since the select function still requires synchronization. We therefore propose probabilistic, synchronization-free algorithms as alternative implementations of the select function, which significantly improve scalability by randomly distributing data structure operations among the k partial data structures. In this case, k-linearizability is only approximated with some probability.

The value of k directly determines the scalability of the data structure, i.e., it determines how many data structure operations can potentially be performed concurrently and in parallel without causing contention. In Scal different types of linearizable data structures can be implemented as k-linearizable data structures. In particular, the synchronization mechanism of the original data structure is orthogonal to Scal. Note that k and the select function can be configured by the programmer at compile time or online with the help of performance counters. For example, a select function may be chosen with $k = 1$ under low contention and with increasing k as contention increases. Scal provides the same interface as the original data structures and allows to express concurrency just in the variable k, which may result in a software engineering benefit. Programmers do not need to worry about complex implementation techniques to improve the scalability of data structures [15].

We claim the following contributions: 1. The notion of k-linearizability and a set of load balancing algorithms for trading off adherence to data structure semantics and scalability. 2. The design of Scal that provides k-linearizability for concurrent data structures using load balancing select functions and partial data structures. 3. The implementation of Scal that consists of various load balancing select functions and data structures which can be arbitrarily combined. 4. An evaluation of Scal and a detailed analysis of its components. The results of the experiments confirm that Scal shows in the presented benchmarks positive scalability for concurrent data structures that typically do not scale under high contention.

The rest of the paper is organized as follows. In Section 2 we introduce the notion of k-linearizability and its properties. In Section 3 we present the generic structure of Scal as well as different select functions and data structures. In Section 4 we discuss related work. Experimental results are presented and discussed in Section 5. The conclusion is in Section 6.

2 k-Linearizability

The definition of k-linearizability is based on the original definitions of sequentiality and linearizability [11]. Sequentiality and linearizability are correctness conditions that determine in which order concurrent operations on a shared data structure may be performed such that each operation appears to take effect instantaneously. We consider data structures such as stacks and queues that provide an insert and a remove operation, and store elements in a given order determined by an ordering condition such as last-in first-out (LIFO), first-in first-out (FIFO), or highest priority first. We focus on stacks and queues here since k-linearizability has an effect on the order in which elements are stored that is monotone in k, i.e., the larger the k the more the elements may be out-of-order, cf. Proposition 1. Monotonicity establishes a bounded relationship between adherence to data structure semantics and k-linearizability and thus scalability. However, note that, similar to the definition of linearizability, our definition of k-linearizability works for any concurrent data structure. In the following we discuss correctness and ordering conditions in more detail and then introduce the definition of k-linearizability.

We use the concept of a history H to model the execution of a concurrent system. A history H is a finite sequence of invocation and response events of operations performed by concurrent threads on a shared data structure. An operation op_0 precedes operation op_1, if the response event of op_0 happens before the invocation of op_1. The sequential order in which a single thread performs operations is called program order. We omit further details and refer the reader for a formal definition of a history as well as for how histories are verified to be sequential or linearizable to [11, 10].

Let Σ be the set of possible states of a shared data structure after applying a history H to an empty instance of the shared data structure using a correctness condition ξ and an ordering condition o. The states in Σ are sequences of elements which may be stored in the data structure instance after performing a series of concurrent operations. Elements in the sequences are ordered from left to right. For example, using ordering condition FIFO an insert operation adds an element to the right end of the sequence and a remove operation removes the left-most element of the sequence. Let function $\Phi : \mathcal{H} \times \Xi \times \Omega \rightarrow \Pi$ return a set $\Sigma \in \Pi$ of possible states for a history $H \in \mathcal{H}$, a correctness condition $\xi \in \Xi$, an ordering condition $o \in \Omega$, where $\mathcal{H}$, Ξ, Ω, and Π are the sets of histories, correctness conditions, ordering conditions, and possible states of a shared data structure, respectively.

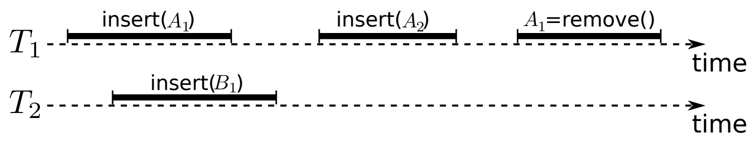


Fig. 1. History H_1

Sequentiality requires operations performed by a single thread on a shared data structure to take effect in program order. The relative order of operations performed by different threads does not need to be kept. A history H is sequential if the first event of H is an invocation that is immediately followed by a matching response, and each response is immediately followed by an invocation (except possibly the last one). Figure 1 shows a history H_1 of two threads performing insert and remove operations on a shared data structure. The possible states of a shared data structure after applying history H_1, correctness condition sequentiality, and ordering condition FIFO to an empty instance of the shared data structure are

$$\Phi(H_1, \mathit{sequentiality}, \mathit{FIFO}) = \{< B_1, A_2 >, < A_2, B_1 >\}.$$

The remove operation of history H_1 returns element A_1 which must therefore be the first element stored in the data structure instance. Elements B_1 and A_2 may be stored in arbitrary order since the order of operations performed by different threads does not need to be kept.

Linearizability requires that each data structure operation takes effect at some time instant between its invocation and response. A history H is linearizable if there is a sequential history S which implements H such that, if operation op_0 precedes operation op_1 in H, then the same is true in S. Hence, operations in a linearizable history which overlap in time may take effect in arbitrary order whereas non-overlapping operations must take effect in sequential order. Let us consider again history H_1 of Figure 1. The insert operations of elements A_1 and B_1 overlap in time and may therefore take effect in arbitrary order whereas the insert operations of elements B_1 and A_2 do not overlap in time and therefore must take effect in sequential order. Thus there is one possible state of a shared data structure after applying history H_1, correctness condition linearizability, and ordering condition FIFO to an empty instance of the shared data structure:

$$\Phi(H_1, \mathit{linearizability}, \mathit{FIFO}) = \{< B_1, A_2 >\}.$$

Next we introduce the notion of virtual delay, which allows us to weaken the original correctness conditions by increasing the number of possibilities to reorder operations. We assume that each operation returns at its response event but may then still be virtually delayed until its so-called virtual response event. The time span between the response event and the virtual response event of an operation is called virtual delay.

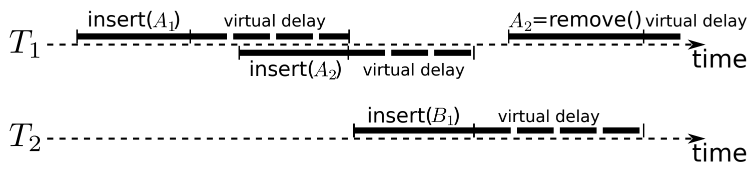


Fig. 2. History H_2 with virtual delay 1

Non-overlapping operations can be extended with virtual delays to become operations that overlap. As a result, the order in which these operations take effect can be changed since they happen virtually in parallel. Note that even operations of a single thread can be extended with virtual delays to overlap, which allows us to change the program order of these operations, as depicted in Figure 2. Using the notion of virtual delay we define k-sequentiality as follows.

Definition 1 (k-Sequentiality) *A history H_k is k-sequential with $k > 0$ if it is obtained from a sequential history H by virtually delaying each response event in H until the $(k-1)$th subsequent response event.*

We also say that a k-sequential history is a history with virtual delay $k-1$. Note that 1-sequentiality is equivalent to sequentiality, i.e., a sequential history is a history with virtual delay 0. The possible states of a shared data structure after applying history H_2, correctness condition 2-sequentiality, and ordering condition FIFO to an empty instance of the shared data structure are

$\Phi(H_2, 2{-}sequentiality, FIFO) = \{<A_1, B_1>, <B_1, A_1>\}$.

The insert operations of elements A_1 and A_2 virtually overlap in time and may therefore take effect in arbitrary order. Element A_2 happens to be the first element returned by the remove operation, so it must be the first element in the data structure. Elements A_1 and B_1 may be stored in arbitrary order in the data structure.

The program order of the operations performed by thread T_1 in H_2 without virtual delay determines that element A_1 must be located in the shared FIFO queue before element A_2, but A_2 is returned by the remove operation before A_1. This implies that the program order is not kept. Hence, history H_2 is not sequential and therefore not linearizable, but it is 2-sequential.

Next k-linearizability is defined according to the original definition of linearizability [11].

Definition 2 (k-linearizability) *A history H is k-linearizable if there is a k-sequential history S which implements H such that, if operation op_0 precedes operation op_1 in H, then the same is true in S.*

We say that a data structure is k-linearizable if all valid histories of its use are k-linearizable. Note that 1-linearizability is equivalent to linearizability. Let us consider again history H_2 of Figure 2. The possible states of a shared data structure after applying history H_2, correctness condition 2-linearizability, and ordering condition FIFO to an empty instance of the shared data structure are:

$\Phi(H_2, 2{-}linearizability, FIFO) = \{< A_1, B_1 >\}$.

The insert operations of elements A_1 and A_2 virtually overlap in time. The insert operations of elements A_1 and B_1 do not overlap in time. Element A_2 is the first element returned by the remove operation. Therefore, element A_1 must be the second element in the data structure followed by element B_1 which results in the data structure state mentioned above.

The following proposition states how different virtual delays affect the number of possible data structure states after applying a given k-linearizable history to a shared data structure.

Proposition 1 *For a given history H, correctness condition k-linearizability, and an ordering condition o, it holds that, for any virtual delays* $k_1 - 1$ *and* $k_2 - 1$ *with* $k_1 > 0$ *and* $k_2 > 0$*, if* $k_1 \leq k_2$ *then* $\Phi(H, k_1{-}linearizability, o) \subseteq \Phi(H, k_2{-}linearizability, o)$.

A larger virtual delay increases the number of possibilities to reorder data structure operations of a given history, which results in a larger number of possible shared data structure states and therefore increasingly weakens shared data structure semantics. For $k_1 = k_2$ Proposition 1 holds trivially since $\Phi(H, k_1{-}linearizability, o) = \Phi(H, k_1{-}linearizability, o)$. If $k_1 < k_2$ and if all operations in history H are overlapping with each other then again $\Phi(H, k_1{-}linearizability, o) = \Phi(H, k_2{-}linearizability, o)$. Otherwise the larger virtual delay of k_2 allows more reordering combinations of operations then k_1 which results in $\Phi(H, k_1{-}linearizability, o) \subset \Phi(H, k_2{-}linearizability, o)$.

3 Scal: *k*-linearizable data structures

Scal implements k-linearizable data structures using a select function and k partial data structures, which are identical instances of a given data structure. The select function determines on which partial data structure an operation is performed. Scal introduces a wrapper that distributes operations on the partial data structures as depicted in the pseudo code in Listing 1.1. The actual data structure operation is given in the parameters of the generic Scal op function, e.g. as a function pointer. In case of an insert operation the parameters contain the given element and the return statement returns a boolean value indicating whether the insert operation was successful. In case of a remove operation an element is returned if the data structure is not empty, otherwise NULL is returned.

In the following we discuss different types of select functions, partial data structures, possible optimizations, and concurrency patterns.

3.1 Select Function

A select function that provides k-linearizability must distribute operations over the k partial data structures evenly. Achieving an even distribution of operations requires

Listing 1.1. Scal generic structure

```
op(data_structure, parameters) {
  partial_ds = select(data_structure);
  return partial_op(partial_ds, parameters);
}
```

expensive global coordination mechanisms, which may anyway provide positive scalability in some cases because more operations may be executed concurrently but may still be too restrictive under high contention. Select functions that approximate k-linearizability probabilistically without global coordination may scale to a larger amount of concurrent load. In this case, the adherence to the given data structure semantics may not be weakened much further as long as the k is sufficiently large and the data structure is sufficiently utilized. In general, the better a select function distributes operations over the partial data structures, the more operations can run concurrently and in parallel, and the better the semantics of the original data structure are approximated. In addition, a select function should be computationally efficient to minimize its overhead.

Perfect Load Balancing A select function that provides a perfect balance of operations and thus k-linearizability can be implemented with global counters that indicate which partial data structure is to be used next. For example, in a FIFO queue two global counters are sufficient. One counter indicates on which partial data structure the last enqueue operation was performed and the other counter indicates on which partial data structure the last dequeue operation was performed. The global counters are accessed and modified using atomic operations, which can cause cache conflicts on high contention when multiple threads modify the same memory locations. Positive scalability can be achieved under low concurrent load since the select function itself is simple and contention on the shared memory locations rarely happens. A perfectly balancing select function provides k-linearizability. We refer to it as perfect select function. Note that it does not provide linearizability for $k = 1$ in Scal since it is not executed atomically with the partial data structure operations.

Randomized Load Balancing Another approach is to use a select function that randomly distributes operations over partial data structures. This approach, also known as randomized load balancing, has been proven to provide good distribution quality if the random numbers are distributed independently and uniformly [2, 3]. However, generating such random numbers may be computationally expensive. Therefore, it is essential to find the right trade-off between quality and overhead of random number generation. An efficient random number generator that produces evenly distributed random numbers was discussed in [17]. The distribution quality of a select function based on a random number generator determines the probability of approximating k-linearizability. We refer to a select function based on a random number generator as random select function.

Suppose that n operations are performed on k partial data structures using a random select function. With a probability of at least $1 - O(\frac{1}{k})$, the maximum number of operations performed on just one partial data structure is $\frac{n}{k} + \Theta(\sqrt{\frac{n \log k}{k}})$ [3]. Thus a larger k leads to a better balance of operations and thus to a higher probability of approximating k-linearizability.

In order to improve the balancing quality of the random select function d partial data structures with $1 < d \leq k$ may be chosen randomly. Out of the d partial data structures the instance that contributes most to a better balance is selected. For example, an enqueue operation on a FIFO queue may be performed on the partial data structure that contains the fewest elements. We refer to such a select function as d-random select function. The overhead of the d-random select function increases linearly in d since the random number generator is called d times. Suppose again that n operations are performed on k partial data structures. With a probability of at least $1 - O(\frac{1}{k})$, the maximum number of operations performed on one partial data structure is then $\frac{n}{k} + \Theta(\frac{\log \log k}{d})$ [3]. The parameter d allows us to trade off balancing quality and global coordination overhead. Moreover, $d = 2$ leads to an exponential improvement in the balancing quality in comparison to the random select function. Note that $d > 2$ further improves the balancing quality but only by a constant factor [3]. Again, a larger k leads to a better balance of operations and thus to a higher probability of approximating k-linearizability.

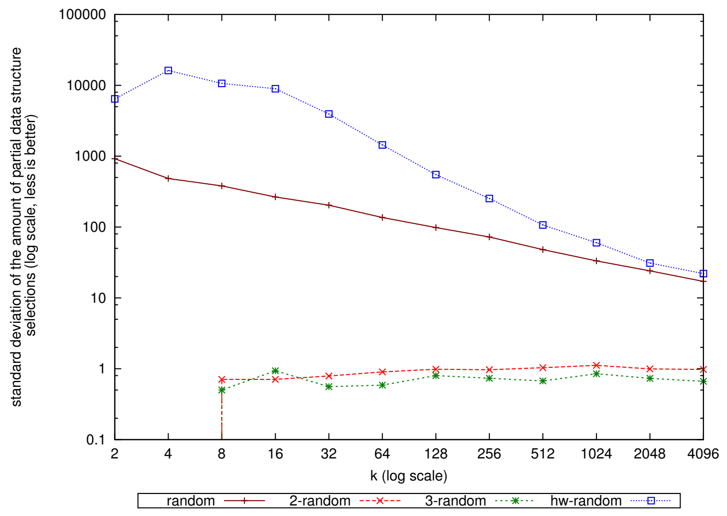


Fig. 3. Balancing quality of different random select functions with increasing number of partial data structures (k)

We conducted several experiments to evaluate the balancing quality of four different random select functions: random, 2-random, and 3-random use a simple but efficient random number generator as discussed in [17]; and hw-random is a random select function that takes the time stamp counter register of the CPU (RDTSC) modulo k. During each experiment the select function is executed a million times and we keep track how often each partial data structure is selected. Figure 3 shows the standard deviation of the amount of partial data structure selections. For each random select function the experiment is repeated using values between 2 and 4096 for k. For example, the standard deviation of how often a partial data structure among 16 partial data structures ($k = 16$) is selected by the 2-random select function is 1 and 10000 by the hw-random select function. hw-random produces the worst distribution among the four evaluated select functions. The experiments also confirm that the d-random select functions provide a significant improvement in balancing quality in comparison to the random select function. The standard deviation of the amount of performed operations on the k partial data structures using the d-random select functions is approximately 1. However, in terms of execution time the d-random select functions are d times slower than the random select function since their random number generator is invoked d times more often.

Thread-based Load Balancing Thread-local data can be used as selection criteria for a select function. For example, a select function can be implemented as a static mapping between thread IDs and partial data structures. All operations of a given thread are forwarded to a unique partial data structure. Another example are thread-local round-robin counters which distribute data structure operations of a single thread in round-robin manner over the partial data structures. Both strategies are efficient since the select function is simple and just operates on thread-local data. However, arguing about distribution quality and probability of reaching a given balance is difficult, since the behavior of all threads operating on the data structure determine the balance and not a global random number generator.

Hardware-based Load Balancing A hardware-based select function can take advantage of CPU-local data structures. For example, if a select function is executed by a given thread on CPU core i it may choose partial data structure i. Similar approaches are already widely used in various operating systems such as in the Linux scheduler and in userland programs such as the mostly lock-free memory allocator [7]. CPU-local data structures increase locality and reduce interconnect traffic. Realizing this in userland requires mechanisms like multi-processor restartable critical sections [7] or scheduler activations [1], which give userland programs information about scheduling decisions. In the future we plan to evaluate hardware-based select functions for userland applications.

3.2 Partial Data Structures

In Scal a data structure is composed of k partial data structures. Each partial data structure is an instance of the same unmodified linearizable data structure and treated as a black box. In particular, a partial data structure uses the same synchronization mechanisms as the original linearizable data structure.

The remove operation of a k-linearizable data structure returns with a probability of $1/k$ the same element as the original linearizable data structure. A larger k increases the probability that more operations are performed concurrently and in parallel. However, it decreases the probability of choosing the partial data structure that approximates the original data structure semantics best, which weakens the Scal approximation of the given data structure semantics.

In the following we discuss how different types of data structures that maintain an order of the stored elements are handled in Scal.

Time-Dependent Data Structures For time-dependent data structures the time instant of the data structure operation determines the order of the elements in the data structure.

The most prominent representatives of this type of data structure are stacks and FIFO queues. They typically provide an insert and a remove operation. The semantical weakening introduced by k-linearizability becomes apparent when performing a remove operation. The remove operation may return an element that is in timely order at most k elements away from the element that would have been returned by the linearizable data structure.

Value-Dependent Data Structures In value-dependent data structures the values of the elements determine the order of the elements in the data structure. Such data structures typically provide an insert and a remove operation with a given ordering condition, e.g., remove the element with the largest value. Prominent representatives of such data structures are priority queues. Note that in order to balance the elements of a priority queue, the d-random select function has to take the values of the elements into account and not just the number of elements in the partial data structures [21, 4].

3.3 Optimizations

Different optimizations can be applied to the generic structure of Scal to improve its applicability and performance. In the following we discuss a backoff algorithm that improves the applicability of Scal and a mechanism to tune Scal online to achieve better scalability.

Backoff Algorithm Some applications are based on the assumption that a remove operation returns an element if there exists at least one element in the data structure, or that an insert operation fails only if the data structure is full. A k-linearizable data structure that does not meet these requirements can lead to deadlocks, crashes, or abnormal behavior of the application. Scal as introduced above does not meet these requirements. For example, the select function of a remove operation may choose an empty partial data structure although there exist not empty ones, or the select function of an insert operation may choose a full partial data structure although there exist not full ones.

In order to correct a bad choice of a select function we propose so-called backoff algorithms. A backoff algorithm can be implemented based on the global state of the data structure or using a heuristic. The global state, for instance, could be represented by a counter that holds the number of elements in all partial data structures. The counter is incremented after a successful insert operation and decremented after a successful

Listing 1.2. Precise backoff algorithm

```
op(data_structure, parameters) {
  do {
    partial_ds = select(data_structure);
    elem = partial_op(partial_ds, parameters);
    if (valid(elem)) {
      update(counter, parameters);
      return elem;
    }
  } while (valid(counter, parameters));

  return null;
}
```

Listing 1.3. Heuristic backoff algorithm

```
op(data_structure, parameters) {
  checks = MAX_CHECKS;
  while (checks != 0) {
    partial_ds = select(data_structure);
    elem = partial_op(partial_ds, parameters);
    if (valid(elem))
      return elem;
    else if (!valid(elem) && checks == 0)
      return null;
    checks--;
  }
}
```

remove operation. If an operation does not succeed the backoff algorithm inspects the counter to determine whether the data structure is full or empty and restarts or aborts the operation, as shown in Listing 1.2. Updating and inspecting the global counter requires synchronization and can lead to cache conflicts, which limits scalability and may deteriorate the overall performance.

A heuristic backoff algorithm may simply retry a certain number of times before deciding that a data structure operation cannot be performed. Listing 1.3 extends the generic Scal wrapper of Listing 1.1 with such a heuristic. In the worst case this heuristic chooses MAX_CHECKS times an improper partial data structure before aborting. The average number of retries depends on different factors such as the data structure operation call rate or the application workload.

We implemented both backoff algorithms discussed above and present experimental results in Section 5.

Self-tuning Scal The number k of partial data structures and the type of select function are tuning parameters of Scal. Both parameters influence the scalability and the semantics of data structures.

In our current implementation of Scal k is set at program startup time. However, k could be adapted online according to profiling information provided by different performance counters. For example, for lock-based data structures the number of failed attempts of taking a lock over a given time period or for lock-free data structures the number of operation retries over a given time period are significant performance indicators. The adaption algorithm for k can be implemented as follows: k can be increased at any point in time. Decreasing k by l partial data structures puts the data structure in a transient state where insert operations are performed on the first $k - l$ partial data structures. Moreover, remove operations may be performed on all k partial data structures. The end of the state transition is reached when the last l partial data structures are empty or k is increased by more than l partial data structures.

As discussed in Section 3.1 different select functions provide different performance benefits depending on the workload. Therefore, it might be beneficial to change the select function at runtime. Self-tuning Scal is future work.

3.4 Concurrency Patterns

In this section we describe common producer-consumer scenarios and how k-linearizable data structures may be configured to achieve scalability. The presented patterns may be useful in a number of applications but are not a complete list of patterns and the proposed configurations may not provide scalability in general.

Static Producer-Consumer Ratio A static producer-consumer ratio exists in applications that use a static number of producers and consumers, which do not change at runtime.

In an application with one producer and n consumers or vice versa, the single producer or consumer instance may use a select function that distributes data structure operations over all partial data structures. The n producers or consumers may then be assigned to a single partial data structure or to a subset of the partial data structures resulting in cache benefits and reduced interconnect traffic.

In a scenario with a fixed number of m producers and n consumers the producers and consumers could be assigned to a single partial data structure or to a subset of partial data structures. A select function that dynamically assigns threads to different partial data structures may then not be needed.

Dynamic Producer-Consumer Ratio A dynamic producer-consumer ratio can be found in applications where the number of producers and/or consumers is not known at application start-up time and can change at runtime.

With a fixed number of producers and a dynamic number of consumers the producers can be assigned to a single partial data structure or to a subset of partial data structures. The consumers may then take advantage of a select function that distributes operations dynamically over all partial data structures. The same holds for the opposite setting with a fixed number of consumers and a dynamic number of producers.

If the number of producers and consumers is dynamic, a select function that distributes operations dynamically over all partial data structures may be used for both producers and consumers.

4 Related Work

In this section we discuss previous work on concurrent systems and concurrent data structures that are related to Scal. We implemented some of the related data structures and evaluated them in experiments in Section 5.

4.1 Concurrent Systems

CPU-local data structures typically help to reduce contention on shared data structures, improve cache utilization, and reduce interconnect traffic. For example, multi-processor thread schedulers, like the completely fair scheduler of the Linux kernel, use CPU-local run queues to eliminate the single linearization point of a global run queue. CPU-local data structures are mostly used in operating system kernels but there are also userland applications that take advantage of this design [7]. Random load balancing between CPU-local run queues of a thread scheduler where each core randomly picks another core to balance tasks between their CPU-local run queues was discussed in [18]. The authors showed that such a distributed random load balancing strategy approximates sufficiently the task order of a scheduler based on a single run queue. A study in the context of earliest-deadline-first (EDF) schedulers was conducted in [5]. A global EDF scheduler with a single run queue was compared to a partitioned EDF scheduler that uses CPU-local run queues. These approaches tolerate a weakening of scheduling semantics in terms of task order to gain performance improvements. Scal is based on the same idea but provides a generic API to trade off adherence of data structure semantics and scalability.

Scal is also related to distributed data structures that spread their data over multiple computers. For instance, the distributed hash table presented in [8] uses a special management unit that handles load distribution, ensures data consistency, and provides linearizability. The authors showed that such a design provides high throughput and allows to handle a high degree of concurrency. In Scal data is distributed over multiple instances of the same data structure to control the trade-off between adherence to data structure semantics and scalability. Note that Scal could in principle distribute partial data structures over different computers. This is left for future work.

Similar to distributed data structures, distributed databases [22] also spread data over multiple computers. Google's BigTable [6] is a special form of a distributed database designed for large-scale distributed systems. BigTable can be used in different kinds of applications while providing scalability. In distributed databases such as BigTable a database management system (DBMS) is in charge of providing data consistency while balancing the load on the distributed database. The select function of Scal can be seen as a simple DBMS, which distributes load over partial data structures.

A compiler that generates a parallel program and gives only a statistical accuracy guarantee on the output is presented in [13]. The generated program does not produce the same output as its sequential version. This approach simplifies the implementation of the compiler and provides more possibilities for parallelization. Similar to Scal, but on a different level, accuracy is traded for performance.

Software transactional memory [20] gained a lot of interest in recent years since it simplifies programming in a parallel environment and promises better scalability than traditional lock-based synchronization. k-linearizable data structures are orthogonal to software transactional memory. They may allow to speed up systems in which software transactional memory is used when data structure operations are distributed over k partial data structures potentially reducing the number of transaction retries.

4.2 Concurrent Data Structures

Concurrent data structures are key components to provide scalability of applications. Traditionally, scalability of concurrent data structures may be increased by using fine-grained synchronization mechanisms which may allow a higher degree of concurrency in comparison to a single global lock.

In [16] the authors proposed a two-lock queue where one lock protects the head and the other lock protects the tail of the queue. The two locks allow one enqueue and one dequeue operation to be performed concurrently. The approach does not scale well when several threads perform enqueue or dequeue operations concurrently. The same paper introduces a lock-free queue, which is implemented as part of the Java concurrency package. Under high concurrent load this algorithm does not scale since all threads apply at least two compare-and-swap operations in the enqueue operation and one compare-and-swap operation in the dequeue operation. For our experiments we implemented the lock-free queue and showed that its Scal implementation scales. Another lock-free queue is the basket queue of Hoffman et al. [12].

Treiber presented a lock-free stack algorithm, which uses a compare-and-swap operation to manipulate the top pointer of a stack [23]. We evaluate this algorithm with different Scal configurations in our experiments and show that it scales with Scal. The performance of the lock-free stack algorithm can be improved by using a so-called elimination backoff stack [9]. It combines complementary data structure operations of threads to minimize global data structure access. A non-linearizable stack algorithm based on the elimination tree technique was proposed in [19]. It is a lock-free algorithm which introduces a high constant overhead. This significantly degrades performance under low concurrent loads. Additionally, it is not linearizable and the authors do not discuss the trade-off between data structure semantics and scalability.

Priority queues based on skip-lists allow fine-grained synchronization but also suffer from synchronization overhead. In our experiments we evaluate the lock-based skip-list algorithm presented in [14] and show that it scales in Scal. A skip-list is composed of a set of sorted linked lists. Each list has a level, ranging from the lowest level 0 to a given maximum level. List level 0 contains all the list elements and each higher-level list is a sublist of the lower-level lists containing links to the lower level. The higher-level lists can be viewed as shortcuts into the lower-level lists. An element in the skip-list is found by starting at the highest-level list. Each level is traversed until the right shortcut to the lower level is found which is used to descend in the level hierarchy. This procedure is performed until list level 0 is reached where the elements are located.

5 Experiments

In this section we discuss the results of our experiments. We evaluate the overhead of Scal and compare different lock-based and lock-free data structures with their *k*-linearizable counterparts.

All experiments ran on a server machine with four 6-core 2.1GHz AMD Opteron processors (24 cores) and 48GB of memory on Linux 2.6.32. In all experiments the benchmark threads were executed with real-time priorities to minimize system jitter. All algorithms are implemented in C and compiled using gcc 4.3.3 with -O3 optimizations. Allocation and deallocation of elements used in the data structures was done on a thread-local basis to minimize cache problems and to avoid scalability issues introduced by the allocator.

5.1 Overhead

select function	no contention	high contention
perfect	51 ns	3113 ns
random	59 ns	64 ns
2-random	108 ns	259 ns

Table 1. Select function overhead

We examine the perfect, random, and 2-random select functions in a no-contention scenario where one thread operates on a given data structure and in a high-contention scenario where 24 threads operate on a given data structure. The average overhead in nanoseconds of the select functions is depicted in Table 1. In the no-contention scenario the perfect select function introduces the lowest overhead with just 51 nanoseconds whereas under high contention it takes an average of 3113 nanoseconds to complete a request. Contention occurs on the global round-robin counters, which are a scalability bottleneck as discussed in Section 3.1. The random select function shows similar performance in the no-contention and in the high-contention scenario. It operates on thread-local data only. The performance of the 2-random select function slightly decreases under high contention because the number of elements in the partial data structures are examined to achieve a better balance of operations. Cache invalidations occur when the counters are accessed and modified by atomic operations of other concurrent data structure operations.

5.2 Producer-Consumer Benchmark

We conducted several experiments using a producer-consumer benchmark that creates high pressure on the shared data structure. The benchmark starts with an empty instance of the data structure. Then, each thread inserts and removes elements in alternating order one million times. In total each thread performs 50% insert and 50% remove operations.

To prevent measuring startup effects we synchronize the start of all threads. Since the data structure is initially empty the maximum number of elements in the data structure is bounded by the number of threads. Moreover, with a linearized data structure each remove operation returns an element because it is always preceded by an insert operation of the same thread. For several different Scal configurations we repeated each benchmark with a different number of threads and measured how much time each thread needed to complete one million operations. Each measurement was done ten times to eliminate measuring inaccuracies. The presented results are the average of the ten measurements. We use the number of operations performed by all threads per millisecond as our metric of throughput. Additionally, we count how often the remove operation returns null as an indicator of how close the semantics of the k-linearizable data structure approximates its linearizable counterpart.

We present two figures for each evaluated data structure. Figure 4 depicts the throughput in operations per millisecond on the y-axis against the number of threads on the x-axis. Figure 5 depicts the semantics using the average number of null-returns per thread on the y-axis (log-scale) against the number of threads on the x-axis.

Throughput The first type of data structure we analyze is a FIFO queue using two different implementations. One implementation uses a lock per partial data structure. The other implementation is the lock-free Michael Scott queue [16].

Figure 4(a) shows the results of the k-linearizable lock-based FIFO queue using different select functions and different values of k. The baseline is the original algorithm not using Scal. If more than one thread is present the baseline's throughput is always lower than of any Scal configuration. The perfect select function does not scale to large numbers of threads and $k > 1$ provides only a constant speed-up. In contrast both random and 2-random select functions achieve positive scalability. The random select function performs slightly better as long as the number of threads is smaller than or equal to the number of available CPUs. For more threads than CPUs 2-random provides better performance and still shows positive scalability. Since a d-random select function provides a better distribution of operations in comparison to the random select function it allows more operations to be performed concurrently.

The throughput of the lock-free Michael Scott queue is depicted in Figure 4(b). Similar to the lock-based FIFO queue the throughput of the baseline decreases if more threads compete for access to the data structure. The throughput of the perfect select function is similar to the lock-based version. For smaller k the random and 2-random select functions perform much better than the lock-based FIFO queue. Moreover, the difference between these two select functions is not as distinct as in the previous implementation.

The results of lock-based and lock-free stack implementations are shown in Figure 4(d) and Figure 4(e), respectively. Similar to the queue experiment the perfect select function does not scale. Both the random and the 2-random select functions provide scalability. The 2-random select function even scales when more threads are started than CPUs are available on the machine. For the lock-free version the random select function provides the best performance and scalability.

The results of the lock-based and lock-free implementations of both the stack and the FIFO queue show that balancing the operations among the partial data structures is more important for the lock-based data structures, especially if more threads are running than CPUs are available on the machine. Descheduled threads that hold the global lock of a partial data structure increase the waiting time of other threads which are interested in the lock. The better the distribution of data structure operations over the partial data structures the less frequent such a scenario happens.

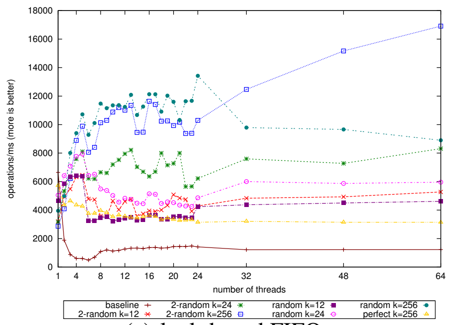
The throughput results of a priority queue implementation are shown in Figure 4(c). Again the perfect select function does not scale. The 2-random select function provides the best performance and scalability since it establishes the best balance of operations. The random selection function scales but performs not as good as the 2-random select function.

Semantics The semantical performance is depicted in Figure 5. We measure the quality of the data structure semantics by counting how often null was returned by a remove operation. This method introduces no measuring artifacts but is not precise. We nevertheless believe that it is a reasonable approach to demonstrate semantical weakening. A precise method would check the distance of the element that is returned by a remove operation to the element that would have been returned by a linearizable version of the data structure. However, such a measurement is extremely costly and would significantly distort the execution of the data structure operations.

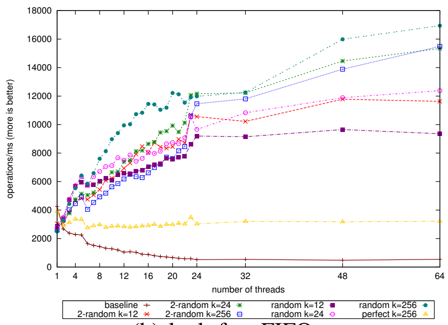
In Figure 5 the baseline is not visible since it is a linearizable data structure and therefore the remove operation never returns null. For Scal these experiments confirm that a larger k leads to more null returns, which indicates that small values of k provide better adherence to the original semantics.

Comparing the different select functions we observed that the remove operation returns null several times when using the random select function whereas the 2-random select function returns null only a few times. The 2-random select function has three orders of magnitude fewer null returns than the random select function. Even though the results of the perfect select function are of an extreme setting with $k = 256$ partial data structures its semantical performance is similar to other select functions that have only 12 partial data structures. We observed that using the perfect select function and a small k the remove operation rarely returns null.

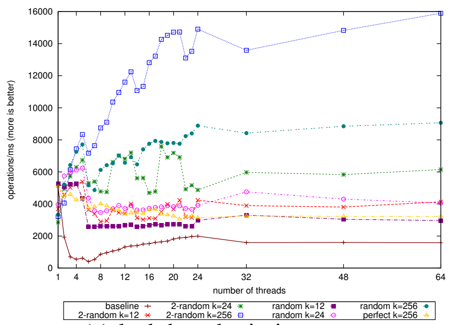
The correlation between balancing operations and scalability as well as semantics is clearly visible. On the one hand, using the random select function the number of times null is returned increases with the number of threads. In the previous section we also saw that its scalability is limited. On the other hand, with the 2-random select function the number of null returns is constant and even slightly decreases if the number of threads increases. It scales better than the random select function. More interestingly, the semantical performance of the 2-random select function is similar and only slightly worse than the perfect select function configuration, while its performance in terms of throughput, however, is much better.



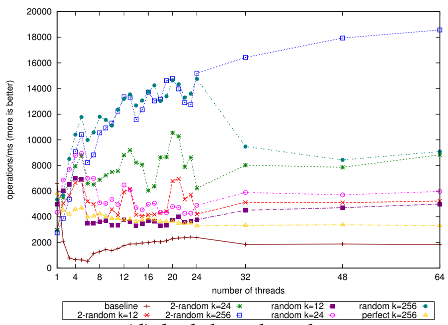
(a) lock-based FIFO



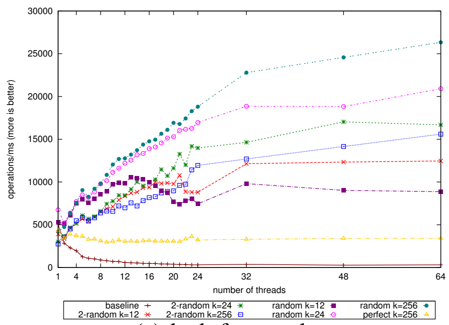
(b) lock-free FIFO



(c) lock-based priority queue



(d) lock-based stack

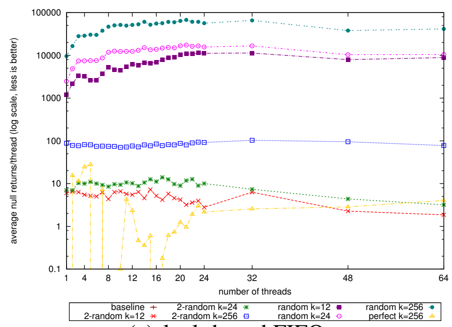


(e) lock-free stack

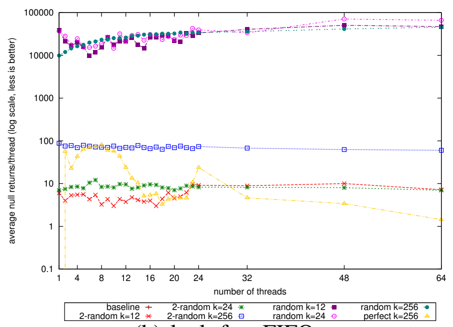
Fig. 4. Throughput in data structure operations/ms with increasing number of threads on a 24-core server machine

5.3 Additional Load

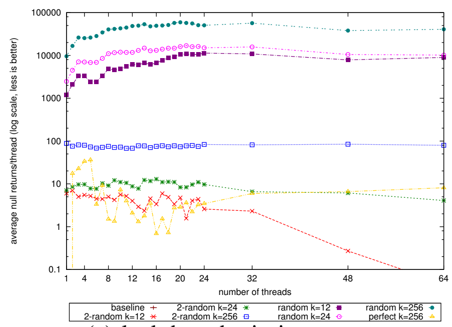
In the producer-consumer benchmarks the threads perform only data structure operations without any computation between each invocation. These benchmarks generate high contention on the shared data structure. In order to have more realistic benchmarks we modified the producer-consumer example such that threads perform some computation between each invocation. The additional load is created by an iterative algorithm



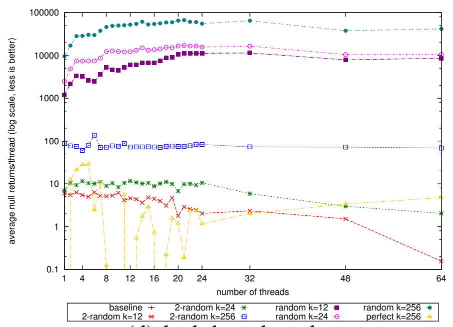
(a) lock-based FIFO



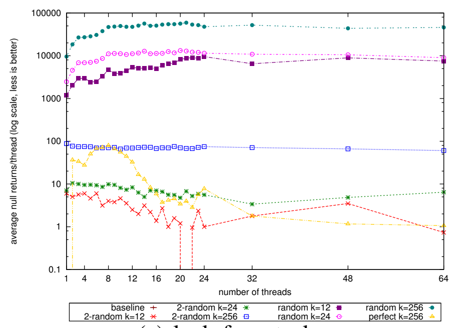
(b) lock-free FIFO



(c) lock-based priority queue



(d) lock-based stack



(e) lock-free stack

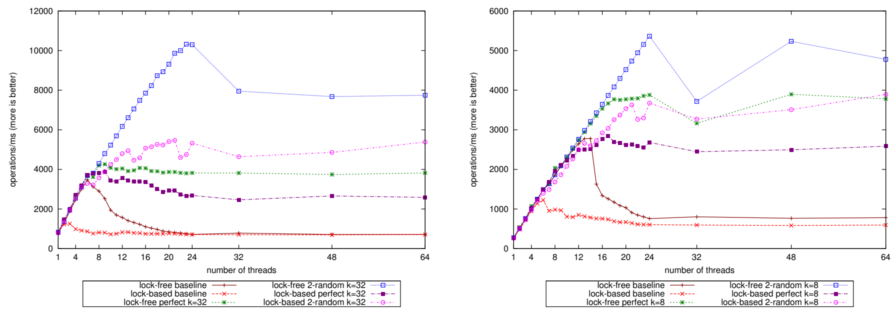
Fig. 5. Adherence to data structure semantics in average null returns/thread with increasing number of threads on a 24-core server machine

that computes π. In the following we present the results of runs using an average computational load of 1130 nanoseconds and 3670 nanoseconds which correspond to 500 and 2000 iterations, respectively. The more load we introduce between the shared data structure operations the less contention is on the shared data structure.

The results of the lock-based and lock-free FIFO queue with additional load are depicted in Figure 6, the results of the stack and priority queue are similar and therefore omitted. The lock-free baseline performs well and scales up to 7 and 14 threads for low and high computational load, respectively. In contrast we can always find a Scal configuration that scales well up to the number of available CPUs. Scalability can be achieved by using a much lower value for k in comparison to the experiments in Section 5.2. For low computational load a configuration $k = 32$ and for high computational load a configuration $k = 8$ is sufficient, as shown in Figure 6(a) and Figure 6(b), respectively. In the low computational load experiment in Figure 6(a) the lock-based and lock-free 2-random select function configuration show the best performance.

In the high computational load experiment, as depicted in Figure 6(b), the lock-free 2-random select function configuration provides the best performance again. The second best configuration is the lock-free perfect select function configuration, which scales up to 19 threads. More importantly, the perfect select function provides better scalability and better performance than the baseline version for either computational load.

The semantical performance of the experiments with computational load is depicted in Figure 7. As discussed in Section 5.2 the perfect select function provides better semantics in comparison to the 2-random select function. In comparison to the previous experiments, which did not have any computational load between each invocation of a data structure operation, the remove operation returns null more frequently. This is expected since the data structure contains less elements due to the additional load and, therefore, the probability of choosing an empty partial data structure is higher than in the previous section.

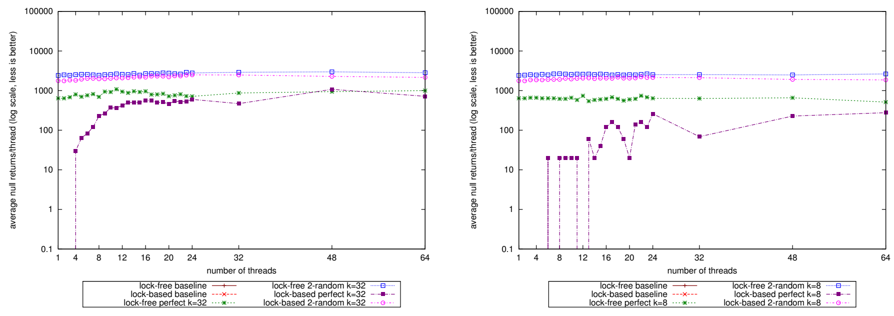


(a) 1130 ns average computational load (b) 3670 ns average computational load

Fig. 6. Throughput of the lock-based and lock-free FIFO queue in operations/ms with additional load in between data structure operations with increasing number of threads on a 24-core server machine

5.4 Backoff Algorithm

In Section 3.3 we introduced a precise and a heuristic backoff algorithm to correct a bad choice of a select function. For example, a bad choice in the producer-consumer



(a) 1130 ns average computational load (b) 3670 ns average computational load

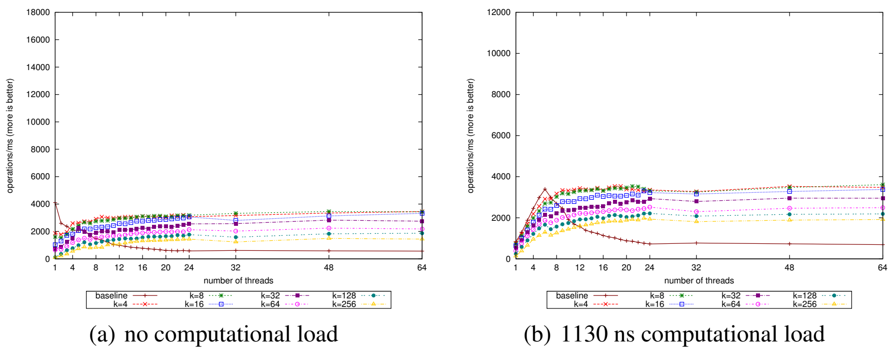
Fig. 7. Adherence to data structure semantics in average null returns/thread of the lock-based and lock-free FIFO queue with additional load in between data structure operations with increasing number of threads on a 24-core server machine

benchmark is when an empty partial data structure is selected for the remove operation. As discussed in Section 3.3 some applications may fail if the data structure operations act differently than expected. We therefore introduced so-called backoff algorithms.

In this section we evaluate the two backoff algorithm for the producer-consumer benchmark of the lock-free FIFO queue using the 2-random select function for three different setups: one with no computational load that produces high contention on the data structure, a setup with a low computational load of 1130 nanoseconds, and a setup with a high computational load of 3670 nanoseconds between each invocation of a data structure operation. The results of the experiments are depicted in Figure 8 and Figure 9. The results of the stack and priority queue are similar and therefore omitted. We only present results of the 2-random select function here since it provides the best trade-off of throughput, scalability, and adherence to the original semantics as discussed in the previous sections. Although the results of the baseline are similar to the numbers presented in the previous two sections we show them in these figures again to better compare and discuss the scalability of the Scal configurations.

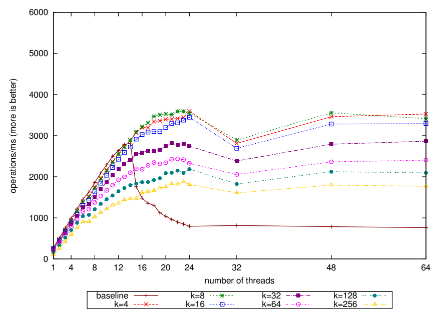
The precise backoff algorithm limits scalability compared to the benchmarks in the previous sections that did not use any backoff algorithm. The speedup of the throughput is only slightly higher than the baseline. However, the throughput stays high, even for a large number of threads compared to the baseline for which the throughput decreases dramatically if more than 7 threads with low computational load (or 14 threads with high computational load) are running. Also note that for these experiments there is an optimal $k = 4$ (or $k = 8$), which is not the largest value of k here. This confirms that a larger k increases the probability of choosing a bad partial data structure, which triggers the backoff algorithm more frequently.

The heuristic backoff algorithm performs much better than the precise backoff algorithm and provides positive scalability, see Figure 9. Its performance is only slightly lower compared to the experiments in the previous sections, which did not use any backoff algorithm. Again, choosing an appropriate value for k, i.e., $k = 16$ is important to achieve positive scalability.



(a) no computational load

(b) 1130 ns computational load

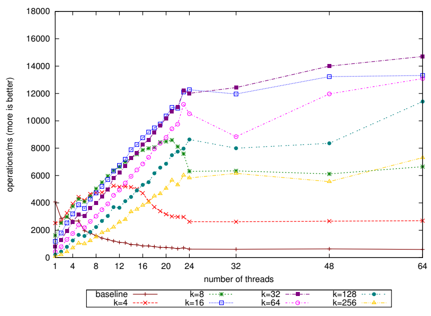


(c) 3670 ns computational load

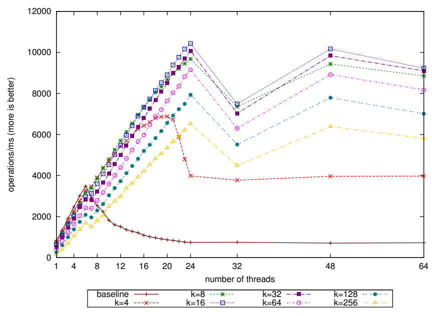
Fig. 8. Throughput of lock-free FIFO queue in operations/ms using the precise backoff algorithm and the 2-random select function on a 24-core server machine

5.5 Summary

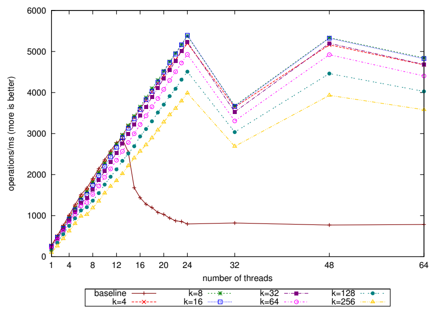
Depending on workload and number of threads, concurrent programs typically generate a call rate of shared data structure operations, which leads to contention above some scalability threshold that in turn depends on the implementation of the shared data structure and the available number and type of CPU cores and their communication infrastructure. With Scal the scalability threshold may actually be changed within the limits of the shared data structure implementation and hardware by appropriately choosing a select function and a number k of partial data structures. A smaller k leads to a lower scalability threshold whereas a larger k raises the scalability threshold. The perfect select function which guarantees k-linearizability allows just a low scalability threshold whereas probabilistic select functions which only approximate k-linearizability allow a larger scalability threshold. A user of Scal needs to take these variables into account



(a) no computational load



(b) 1130 ns computational load



(c) 3670 ns computational load

Fig. 9. Throughput of lock-free FIFO queue in operations/ms using the heuristic backoff algorithm and the 2-random select function on a 24-core server machine

and may choose the configuration with the smallest *k* value and the most accurate select function to provide the best adherence to data structure semantics while still providing scalability.

6 Conclusion

We have presented Scal, an open-source framework for implementing *k*-linearizable approximations of concurrent data structures which enables trading off adherence to data structure semantics and scalability. The implementation is based on so-called select functions and partial data structures. Given a concurrent data structure, Scal relates the adherence to its semantics and its scalability through *k*-linearizability. With sufficiently large *k* and sufficiently utilized data structures even larger degrees of scalability without weakening semantics much further can be obtained by approximating *k*-linearizability probabilistically.

The trade-off between adherence to data structure semantics and scalability is discussed in experiments. Moreover, different select functions and partial data structures are evaluated under different workloads. The experiments confirm that Scal shows in the presented benchmarks positive scalability for concurrent data structures that typically do not scale under high contention.

There are several open questions for future work. In the context of distributed data structures [8] it would be interesting to evaluate how *k*-linearizable data structures perform in distributed systems. Moreover, we plan to apply *k*-linearizability to more complicated data structures and evaluate its applicability and performance. As outlined in Section 3.3 we are going to implement a mechanism that adjusts the number *k* of partial data structures and the type of select function online. Both parameters influence the scalability and the semantics of a given shared data structure. A self-tuning algorithm may thus provide the best performance for a given workload.

References

1. Anderson, T., Bershad, B., Lazowska, E., and Levy, H. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proc. Symposium on Operating Systems Principles (SOSP)* (1991), ACM, pp. 95–109.
2. Azar, Y., Broder, A. Z., Karlin, A. R., and Upfal, E. Balanced allocations (extended abstract). In *Proc. Symposium on Theory of computing (STOC)* (1994), ACM, pp. 593–602.
3. Berenbrink, P., Czumaj, A., Steger, A., and Vöcking, B. Balanced allocations: The heavily loaded case. *SIAM Journal on Computing 35*, 6 (2006), 1350–1385.
4. Berenbrink, P., Friedetzky, T., Hu, Z., and Martin, R. On weighted balls-into-bins games. *Theoretical Computer Science 409*, 3 (2008), 511–520.
5. Brandenburg, B., Calandrino, J., and Anderson, J. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proc. Real-Time Systems Symposium (RTSS)* (2008), IEEE, pp. 157–169.
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: a distributed storage system for structured data. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)* (2006), USENIX, pp. 205–218.
7. Dice, D., and Garthwaite, A. Mostly lock-free malloc. In *Proc. International Symposium on Memory Management (ISMM)* (2002), ACM, pp. 163–174.
8. Gribble, S., Brewer, E., Hellerstein, J., and Culler, D. Scalable, distributed data structures for internet service construction. In *Proc. ymposium on Operating System Design & Implementation (OSDI)* (2000), USENIX, pp. 319–332.
9. Hendler, D., Shavit, N., and Yerushalmi, L. A scalable lock-free stack algorithm. In *Proc. Symposium on Parallelism in algorithms and architectures (SPAA)* (2004), ACM, pp. 206–215.
10. Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.
11. Herlihy, M., and Wing, J. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.
12. Hoffmann, M., Shalev, O., and Shavit, N. The basket queue. In *Proc. Conference On Principle Of Distributed Systems (OPODIS)* (2007), Springer, pp. 401–414.
13. Kim, D., Misailovic, S., and Rinard, M. Automatic parallelization with statistical accuracy bounds. Tech. Rep. 007, MIT CSAIL, 2010.
14. Lotan, I., and Shavit, N. Skiplist-based concurrent priority queues. In *Proc. International Symposium on Parallel and Distributed Processing (IPDPS)* (2000), IEEE, pp. 263–568.
15. McKenney, P., Gupta, M., Michael, M., Howard, P., Triplett, J., and Walpole, J. Is parallel programming hard, and if so, why? Tech. Rep. 09-02, Portland State University, 2009.
16. Michael, M., and Scott, M. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Symposium on Principles of Distributed Computing (PODC)* (1996), ACM, pp. 267–275.
17. Park, S., and Miller, K. Random number generators: good ones are hard to find. *Communications of the ACM 31*, 10 (1988), 1192–1201.
18. Rudolph, L., Slivkin-Allalouf, M., and Upfal, E. A simple load balancing scheme for task allocation in parallel machines. In *Proc. Symposium on Parallel Algorithms and Architectures (SPAA)* (1991), ACM, pp. 237–245.

19. SHAVIT, N., AND TOUITOU, D. Elimination trees and the construction of pools and stacks: preliminary version. In *Proc. Symposium on Parallel Algorithms and Architectures (SPAA)* (1995), ACM, pp. 54–63.
20. SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proc. Principles of Distributed Computing (PODC)* (1995), ACM, pp. 204–213.
21. TALWAR, K., AND WIEDER, U. Balanced allocations: the weighted case. In *Proc. Symposium on Theory of computing (STOC)* (2007), ACM, pp. 256–265.
22. TAMER, O., AND VALDURIEZ, P. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, 1999.
23. TREIBER, R. Systems programming: Coping with parallelism. Tech. Rep. RJ5118, IBM Almaden Research Center, April 1986.