More Scalable Ordered Set for ETS Using Adaptation

Konstantinos Sagonas Kjell Winblad
Department of Information Technology, Uppsala University, Sweden
firstname.lastname@it.uu.se

Abstract

The Erlang Term Storage (ETS) is a key component of the runtime system and standard library of Erlang/OTP. In particular, on big multicores, the performance of many applications that use ETS as a shared key-value store heavily depends on the scalability of ETS. In this work, we investigate an alternative implementation for the ETS table type `ordered_set` based on a contention adapting search tree. The new implementation performs many times better than the current one in contended scenarios and scales better than the ETS table types implemented using hashing and fine-grained locking when several processor chips are used. We evaluate the new implementation with a set of experiments that show its scalability in relation to the current ETS implementation as well as its low sequential overhead.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Parallel programming; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed and parallel languages; Applicative (functional) languages

Keywords concurrent data structures; search trees; Erlang

1. Introduction

Erlang is a concurrent programming language which is often advertized as supporting "shared-nothing concurrency". Processes in Erlang are very lightweight, they are implemented by the virtual machine instead of being mapped to operating system threads and indeed share no memory by default. However, the main implementation of the language, the Erlang/OTP system, comes with mechanisms that allow processes to share memory that other processes can read and update concurrently. Chief among them is the Erlang Term Storage (ETS), a library which provides a key-value store in-memory database that is implemented in C for efficiency. In many Erlang applications, ETS is heavily used and is a critical component of their implementation.

To benefit from multicores, the Erlang/OTP virtual machine has been adapted to use multiple schedulers which allow Erlang processes to run in parallel. As is well-known however, regardless of how many processes are runnable at any point in an application, the speedup that one can gain from the available hardware parallelism is limited by the sequential parts of the program and the bottlenecks that might exist in the runtime system.

Erlang '14, September 05, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2633448.2633455

Over recent years, the ETS implementation has gradually improved in scalability and performance [12]. For example, some ETS tables are implemented using fine-grained locking and nowadays the user can set various concurrency options [12]. However, as we will soon show in this paper, ETS tables of type `ordered_set` are still a scalability bottleneck when there are concurrent write operations. This is not suprising given that their implementation is based on a data structure protected by a single readers-writer lock.

To ameliorate the situation, in this paper we propose an alternative implementation for ETS tables of type `ordered_set` that, with only a small sequential overhead, allows update operations to take place in parallel, thereby improving the scalability of the system. Our alternative ordered set implementation is a novel concurrent data structure called *contention adapting tree* (*CA tree* for short).

Overview The next section contains a description of the ETS implementation as well as an evaluation of its current scalability. It serves both as background material and presents the motivation for this work. A high-level description of CA trees is given in Section 3. How to properly integrate CA trees into the ETS implementation is described in Section 4. The performance and scalability of CA trees is measured in Section 5 by comparing them to those of the current ETS implementation. This paper ends by related (Section 6) and future work (Section 7) and some concluding remarks (Section 8).

2. Erlang Term Storage

In this section, we will briefly present aspects of ETS that are necessary to understand this paper. For more information about ETS, the evolution of its implementation, and its performance and scalability across Erlang/OTP releases refer to a paper [12] presenting these subjects in detail.

2.1 Table Types and Operations

ETS tables are key-value stores: they store Erlang tuples where one of the positions in the tuple serves as the lookup key. ETS tables come in many flavors. When creating an ETS table, one can specify several options. One of them is `public`, which specifies that all Erlang processes can access this table to read and update its contents concurrently. In this paper, we focus on this kind of ETS tables. Another set of options specifies the *type* of the table, which can be either `set`, `bag`, `duplicate_bag` or `ordered_set`. In tables of type `set` and `ordered_set` all keys are unique. The difference between them is that tables of type `ordered_set` provide traversal in key order, while those of type `set` only provide unordered traversal. The functions for traversal are `first/1`, `last/1`, `next/2` and `prev/2`. Functions `first/1` and `last/1` return the first and last key of a table respectively. The `next/2` function is given a table and a key and returns the key of the next element in the table's order (the key order in tables of type `ordered_set`). The `prev/2` function works symmetrically. To perform safe traversal of a `public` table without an order, programs have to first call the ETS function `safe_fixtable/1`. This will fix the internal structure of the table

to avoid re-orderings of the elements internally. Unfortunately, this can also cause performance degradation if the table changes size while it is fixated.

For completeness, we mention that ETS tables of type `bag` allow the presence of several tuples with the same key, while those of type `duplicate_bag` even allow duplicated tuples. Besides the operations mentioned above, the ETS API supports a large number of other operations. For example, it includes high-level functions like `foldl/3` and `foldr/3` and functions to atomically insert a bunch of elements or atomically update all elements in a list.

2.2 Implementation Aspects

Currently, tables of type `set`, `bag` and `duplicate_bag` are implemented using linear hashing [15] and tables of type `ordered_set` are implemented with AVL trees [1].

By default, every `public` table is protected by a single readers-writer lock. When creating such a shared table one can also specify the performance tuning options `read_concurrency` and `write_concurrency`. The option `read_concurrency` enables an optimization on the table lock for many concurrent read operations. A little bit oversimplified, the read optimized table lock has one memory area for every scheduler in the Erlang virtual machine. These memory areas are called *reader groups*. Every scheduler is assigned to one of these reader groups and uses it to synchronize with write operations. On multicores, this makes it possible for read operations to access the table without any memory contention. When write operations enter the picture, the reader groups can cause some overhead since writers have to read all reader groups. The option `write_concurrency` is supposed to optimize the table for many concurrent write operations. When enabling `write_concurrency` on a hash-based table, fine-grained (i.e. bucket-level) locking is enabled for the data structure. The hash-based tables can thus scale well in the presence of concurrent write operations. The `ordered_set` table type is however unaffected by the `write_concurrency` option. Currently, the ETS documentation does not mention this fact.

2.3 Current Scalability

Next, we will examine the scalability of the current implementation of ETS by varying the table options and operations used. Before describing the benchmark and the hardware setting we will employ, let us mention that the implementation of all tables based on hashing is shared, i.e. they use the same code, and consequently their performance and scalability characteristics are very similar. Therefore, henceforth we will use the `set` type as a representative for all hash-based table types, and we will only include tables of type `set` and `ordered_set` in our benchmarking.

Benchmark Description In this section and in Section 5 we will employ the `ets_bench` benchmark from the BenchErl benchmark suite [5] to measure the scalability of ETS. `ets_bench` is a configurable benchmark designed to measure the scalability of different ETS table configurations. The benchmark measures three phases, one for the `insert` operation, one for the `delete` operation, and one for mixed operations (a mix of `insert`s, `delete`s and `lookup`s). The insert phase performs I `insert` operations with keys randomly chosen from the range $R = [1, K]$. The mixed phase is started with the keys that were inserted during the insert phase and executes M operations with keys randomly chosen from R, where the operations are selected randomly so that $L\%$ of the operations are `lookup`s and $(100 - L)\%$ are updates. The update operations are selected so that half of them are `insert` and the other half are `delete` operations; i.e. so that the table size stays roughly constant during this phase. Finally, the delete phase starts with the keys that exist in the table after the mixed phase and executes I `delete` operations with keys randomly chosen from R. The key range that is used in the benchmark is $[1, 2^{21}]$, the insert phase performs 2^{20} operations and the mixed phase performs 2^{22} operations. The operations performed by a phase are distributed to the worker processes that are set up to run the benchmark so that all processes perform roughly the same number of operations. One worker process is started for every scheduler in the Erlang runtime system. The scalability of the system is thus measured by varying the number of schedulers that the Erlang/OTP system is started with. To ensure reliable results we measure every configuration three times and show the arithmetic mean of these measurements in the graphs. An error bar showing the minimum and maximum measurements is also displayed when the measurements have enough difference to make the error bar visible. We mention in passing that our previous paper on the scalability of ETS [12] also used `ets_bench`, where more information about the benchmark can be found.

Benchmark Setting The machine we use has four Intel(R) Xeon(R) CPU E5-4650 CPUs (2.70GHz), eight cores each (i.e. the machine has a total of 32 physical cores, each with hyperthreading, which makes a total of 64 logical cores). The machine has 128GB of RAM and is running Debian Linux 3.10.17-amd64.

In all benchmark runs we pin the scheduler threads of the VM to hardware threads using the `+sbt nnts` option of `erl`. This option will make the Erlang runtime system first occupy all physical cores on one NUMA node (processor chip), then all logical cores (hyperthreads) on the same NUMA node, then the next NUMA node is filled in the same way, and so on. This way the first sixteen scheduler counts in all graphs show the scalability on a single processor chip. All the code that we are benchmarking is based on Erlang/OTP release 17.0.

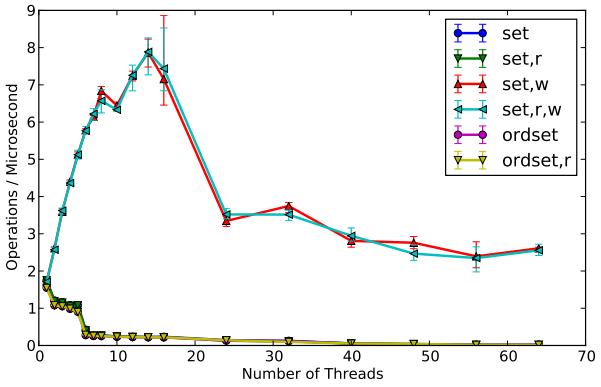
Benchmark Results In the graphs of this section, we use tables of type `set` as representative of the hash-based ETS tables in order to compare them with tables of type `ordered_set` which is what we focus on in this paper. We also enable all combinations of concurrency options that are available for tables of type `set` and `ordered_set`. In particular, an `r` in a graph's label indicates that `read_concurrency` is activated and a `w` indicates that `write_concurrency` is activated. Activating both is indicated with `r,w`. In all graphs we measure scalability by throughput: the number of operations per microsecond that get executed as the number of VM schedulers (hardware threads) increases.

The first set of benchmarks measures the scalability of the basic ETS operations (`insert`, `delete` and `lookup`) in isolation. Figure 1 shows the results. Various conclusions can be drown immediately.

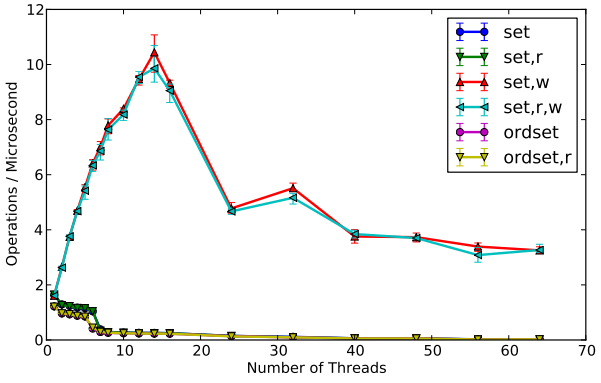
For the update operations (Figures 1a and 1b) the scalability is good only if one activates the `write_concurrency` option, which currently is effective only on tables of type `set`. With this option activated, the throughput of update operations increases when adding cores on a single chip, then it increases but at a slower pace when also adding the hyperthreads on the same chip, and starts dropping once schedulers are on different NUMA nodes.

The `lookup` operation shows a different picture. A readers-writer lock allows many concurrent readers so there is increased throughput in all cases as long as we stay on the same NUMA node. Even on more than one NUMA node, the scalability is pretty good on both `set` and `ordered_set` when the `read_concurrency` option is activated.

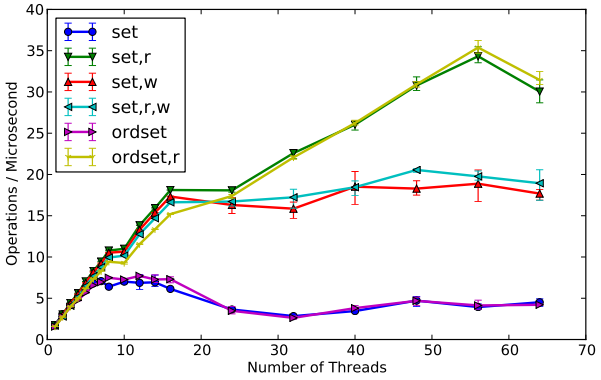
To summarize Figure 1, ETS tables of type `ordered_set` have very bad scalability in update-only scenarios, but perform quite well when the operations are just `lookup`s. Given that in many applications we can expect that the `lookup`s vastly outnumber the updates, one may argue that the scalability of the `ordered_set` table type is typically not so bad in practice if one is careful to enable the `read_concurrency` option. Unfortunately, as we will soon show, even a small amount of update operations can have devastating effects to the scalability of `ordered_set` ETS tables.



(a) `insert`

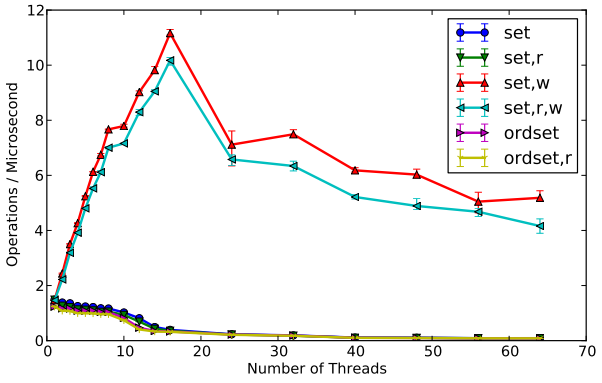


(b) `delete`

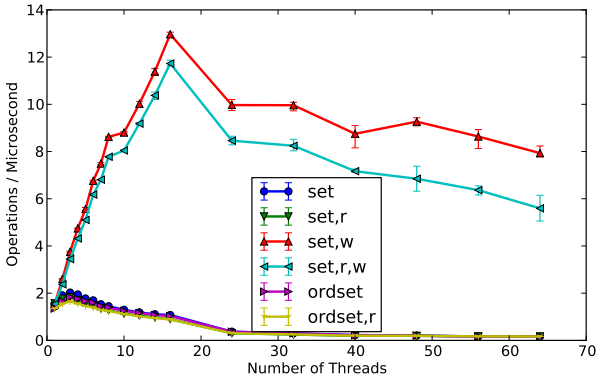


(c) `lookup`

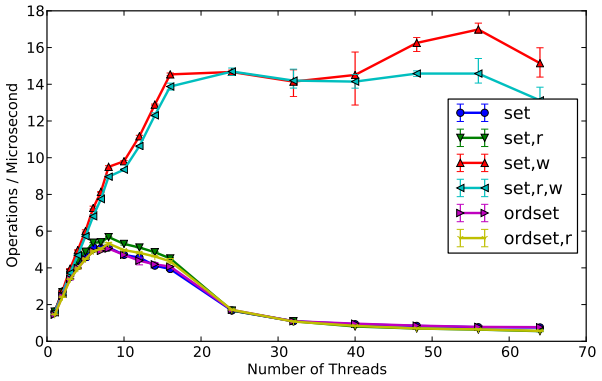
Figure 1: Scalability of ETS table configurations on `insert`, `delete` and `lookup` operations.



(a) 50% Updates, 50% Lookups



(b) 20% Updates, 80% Lookups



(c) 1% Updates, 99% Lookups

Figure 2: Scalability of ETS table configurations using a mix of update and lookup operations.

Figure 2 shows results from running `ets_bench` with configurations where the percentage of `lookups` varies from 50% to 99%. As can be clearly seen in all graphs, the scalability of `ordered_set` is pretty bad even when the amount of updates is very low. Only tables of type `set` with `write_concurrency` enabled show good scalability in scenarios with a mix of operations. Finally, in the current ETS implementation, in scenarios with a mix of operations, one cannot expect increased scalability once schedulers are not on the same NUMA node.

But one thing is clear. Something needs to be done about tables of type `ordered_set` in the presence of updates. In the next sections we will describe an alternative data structure for tables of this type and we will see whether and how much it can improve the scalability of ETS, both compared to the current implementation of `ordered_set` and in comparison to tables of type `set`.

3. A High-Level View of CA Trees

This section provides a bird's eye view of *contention adapting search trees* or *CA trees* for short. A CA tree implements the abstract data type ordered set and supports normal set operations such as insert, delete and lookup as well as efficient ordered traversal. Pseudocode for the implementation of CA trees, their properties, and more information about them can be found in a companion technical

report [21]. Here, we only briefly describe the components of CA trees and give a rough idea about how to assemble them into an efficient implementation. The two main components of CA trees are mutual exclusion locks, which collect statistics about how contended each lock is, and a sequential ordered set data structure that supports split and join operations.

3.1 Components

For a data structure to adapt according to how contended it is, one needs to record the contention of its accesses. Adaptation can occur when enough contention has been detected to justify an adaptation. Likewise, one needs to collect information about lack of contention to be able to adjust accordingly. A natural place to collect such statistics for a data structure containing mutual exclusion locks is in the locks themselves. A straightforward implementation is to increment a counter when a thread needs to wait for a lock and decrement the counter when a thread aquires the lock without waiting. With this statistics available, the data structure can be adjusted when the statistics counter reaches certain thresholds. We define a *statistics collecting lock* as a lock object that provides the usual operations `lock`, `unlock` and `try_lock` as well as a statistics variable that gives an indication of the contention level of the lock.

The second component that CA trees use is a sequential ordered set data structure. Essentially, any such data structure could be used. However, to create an efficient CA tree implementation it is important to consider the efficiency of its operations since the efficiency of a CA tree will in turn depend on these operations. A CA tree adapts to contention by splitting the sequential ordered set data structure in two. The `split` operation creates two new data structures where all keys in one of the resulting parts are smaller than the keys in the other part. To adapt fast to contention the sequential data structure needs to have efficient support for the `split` operation. To adapt to lack of contention we also need an efficient `join` operation. This operation takes two sequential data structures where all keys in one of them are strictly smaller than the keys in the other and creates a new data structure containing the keys of both.

3.2 Assembling the Components

Given a statistics collecting lock and a sequential ordered set data structure one can assemble a CA tree. A CA tree consists of routing nodes and base nodes. Routing nodes contain a routing key, an ordinary mutex lock, a valid flag, one pointer for a left branch and one pointer for a right branch. All keys in the left branch are smaller than the routing key and all keys in the right branch are greater or equal to the routing key. A branch can either be another routing node or a base node. The lock and the valid flag in a routing node are only needed when adapting to low contention by joining trees, an operation which is expected to happen only infrequently. A base node contains a statistics collecting lock that needs to be acquired to access the rest of the data in the base node. Additionally, the base node contains a sequential ordered set data structure and a valid flag.

Figure 3 depicts the structure of a CA tree. The oval shapes are routing notes; the rectangular shapes are base nodes, and the triangular shapes are sequential ordered set data structures. Nodes marked with a valid symbol (a green curve in the figure) are valid, while the node marked with an invalid symbol is no longer in the tree. The search for a key in the tree happens as in a normal binary search tree. To access the content of a base node the statistics lock in the base node needs to be acquired. After taking the statistics lock the valid flag needs to be checked. If this flag is set to invalid, the base node is no longer in the tree and the operation needs to be retried from the root of the tree.

Adapting to high contention works by splitting a base node into two base nodes that are linked together with a new routing node. Before unlocking the original base node, the node needs to be marked invalid so that threads that have been waiting for the lock during the split will see that they need to retry the operation.

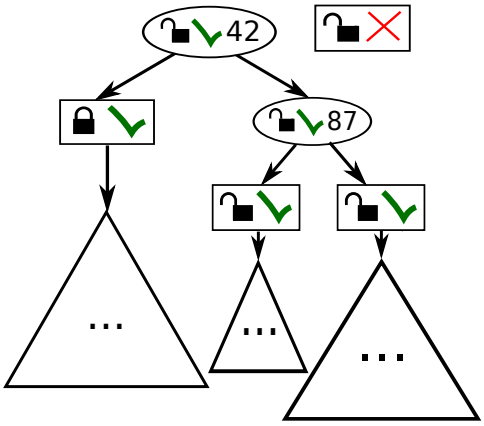


Figure 3: The structure of a contention adapting search tree.

Adapting to low contention is slightly more complicated than adapting to high contention. The low contention adaptation works by joining two neighboring base nodes in the tree. One also has to remove one routing node from the tree. To do this without risking to lose part of the tree, we need to lock the parent of the base node that will be deleted as well as the grandparent (if the grandparent is not the root of the tree). Similarly to the case when adaptation requires a split, both joined base nodes need to be marked invalid before they are unlocked so that waiting threads will retry the operation.

4. Integrating CA Trees into ETS

Two variants of the CA tree have been implemented and integrated into ETS as two new table types for testing purposes. One of these variants uses the *Treap* data structure [3] as the sequential ordered set component and the other one uses an AVL tree [1]. The implementation of the latter is based on the AVL tree code currently used by Erlang/OTP for `ordered_set`.

The CA tree integration is currently a prototype in the sense that it does not yet support the full interface of ETS. The operations currently supported are `insert`, `delete`, and `lookup`. However, extending the implementations to support the full ETS interface is quite easy. The operations that operate on a single key can just be forwarded to the sequential data structure. In the case of the CA tree implementation which is based on the AVL tree, the code for the `ordered_set` implementation can be reused as is.

Operations that atomically operate on several keys can be implemented as they are currently implemented in the `set` table type when fine-grained locking (i.e. `write_concurrency`) is activated. With fine-grained locking enabled, the operations first acquire the table's readers-writer lock in read mode and then the fine-grained lock for the hash table bucket that they need. Operations that atomically operate on several keys take the table lock in write mode and will thus lock out all other operations from the table.

An interesting operation is the one that returns the size of an ETS table. This is what `ets:info(Tab,size)` does. Also, the size of a table is part of the return value of `ets:info/1`. The operation of returning the size of an ETS table can be implemented by using one global counter for each table which is atomically incremented and decremented when elements are added and removed. Actually, this is how returning the size of an ETS table is currently implemented in Erlang/OTP 17.0. When calls to `ets:info(Tab,size)` are frequent, this implementation is very good since retrieving the size

of an ETS table amounts to just a load instruction. However, a global counter is a scalability bottleneck. An alternative implementation for the CA tree would be to keep a counter in every base node. To return the size of an ETS table, the operation would then have to acquire the table lock and sum the size counters of all base nodes. The cost for computing the size would then depend on the contention level, but the scalability bottleneck of using a global counter would be avoided, so this implementation is probably better when `ets:info(Tab,size)` is expected to be called only occasionally.

Base nodes and routing nodes can be removed from the tree by one thread while other threads still have references to them. Thus, these nodes *cannot* be reclaimed with an ordinary `free` call. There exist several ways to deallocate these nodes in a safe way [9, 11, 17]. Our current implementation deals with the reclamation of base and routing nodes by putting them in a free list array located in the table data structure. Once the free list gets full the table is write-locked and unlocked so that no threads can reference elements in the free list. At this point the free list can be emptied and all its elements can be deallocated. This is a form of *quiescent-state-based reclamation* [4, 16]. The table lock is only acquired when the free list is full and deletions of base and routing nodes happens infrequently, so the cost of acquiring the table lock is amortized over many operations. If a CA tree implementation is going to be integrated into the Erlang/OTP distribution, a better way to deal with memory reclamation would be to use the memory reclamation system for lock-free data structures that is already part of the Erlang/OTP code base.

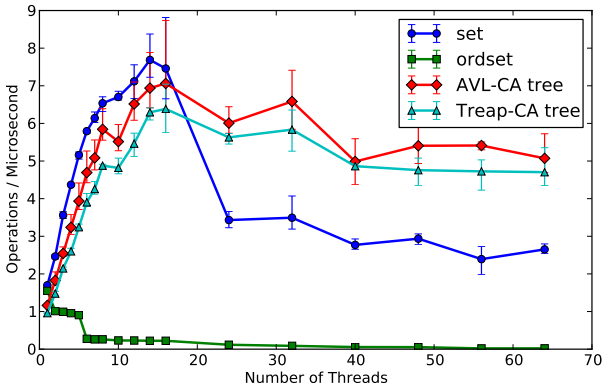
5. Performance Evaluation

This section contains performance and scalability evaluations for the new table types (AVL-based and Treap-based CA trees). We compare the new table types with the current implementations of `ordered_set` and `set`. Tables of type `ordered_set` have `read_concurrency` activated and those of type `set` have both `write_concurrency` and `read_concurrency` activated. As we have shown in Section 2.3, it does not matter much which concurrency options are activated for `ordered_set` when there are update operations, and the difference between activating both `read_concurrency` and `write_concurrency` versus only `write_concurrency` is not large for the `set` type.

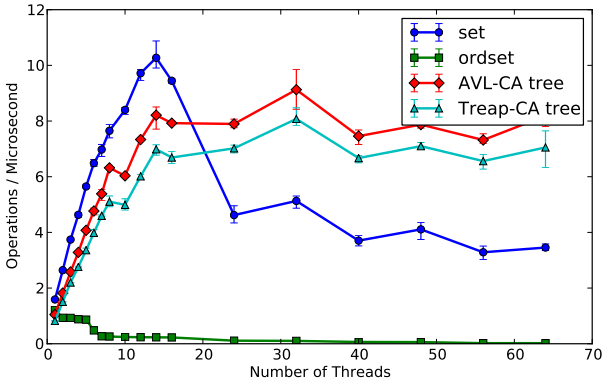
5.1 Scalability w.r.t. Operations and Contention Levels

Figure 4 shows the performance of the table types with varying distributions of operations and varying contention levels. The measurements are collected by `ets_bench` that we described in Section 2.3. The benchmark settings are also the same as in Section 2.3. Both the insert and the delete phase thus do 2^{21} operations each. The lookup and mixed scenarios operate on a set with roughly 2^{21} elements.

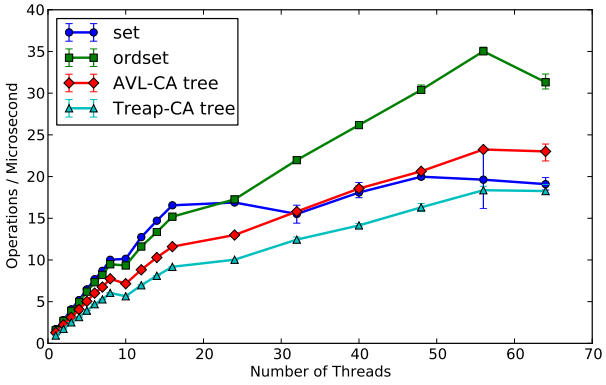
We start by analyzing the insert and delete phases (Figures 4a and 4b). In both these scenarios, the data structures with fine grained locking (`set`, AVL-CA tree and Treap-CA tree) seem to scale relatively well when just one processor chip is used (up to sixteen threads). The scalability deteriorates when using more than one NUMA node for all data structures. However, the performance seems to degradate most for `set`. Memory transfer is much more expensive between NUMA nodes than within a single processor chip which explains this decrease in performance. As we have discussed in our previous paper about the scalability of ETS [12], the `set` table type has a few contended hot spots. For example, only one thread can change the size of a table and there are atomically modified counters for the number of active buckets as well as the size of the table. The memory management functions used by ETS also contain a contended hot spot. The size of the memory consumed by the tables is stored in an integer modified by atomic instructions. This



(a) `insert`



(b) `delete`

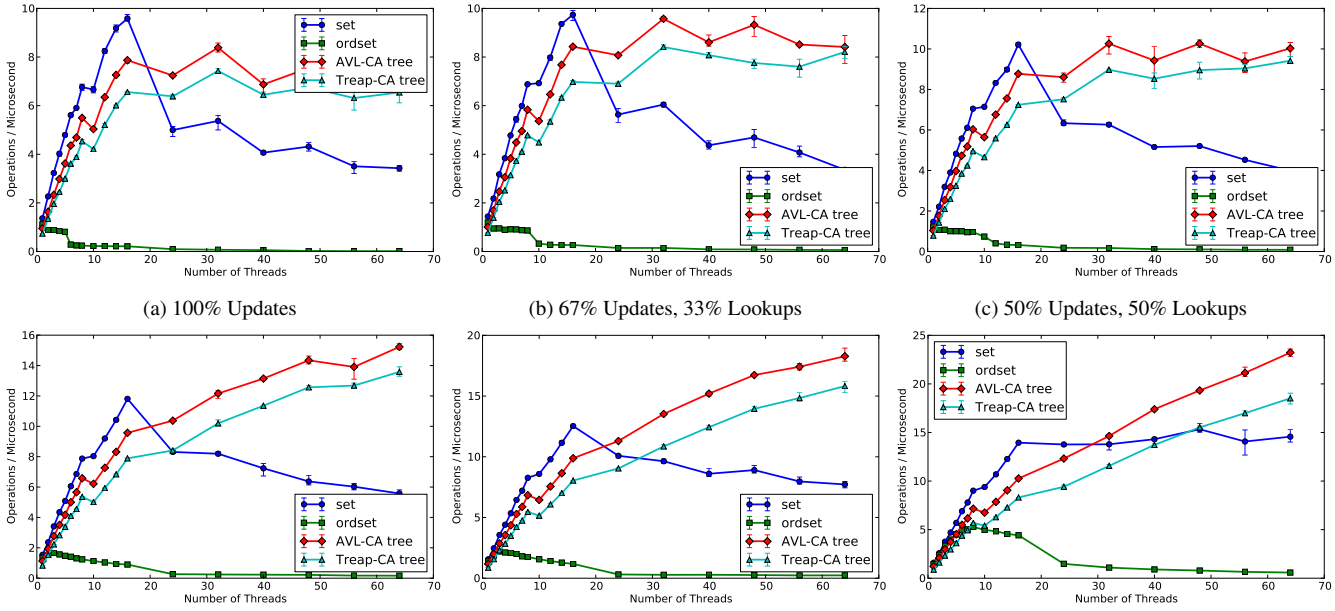


(c) `lookup`

Figure 4: Scalability of the CA tree variants compared to `ordered_set` and `set` with concurrency options activated.

counter is modified every time elements are added to or deleted from the table. Since the new table types also use the same ETS memory management code their performance is also effected by this component. However, the new table types have less hot spots than the `set` and can thus perform better when several NUMA nodes are used.

In the lookup-only case (Figure 4c) just protecting a table with a frequent read optimized readers-writer lock scales best. It is surprising that the `set` with both `read_concurrency` and `write_concurrency` scales so poorly in the read-only case. We suspect that `set` has some problems with false sharing or something



(a) 100% Updates (b) 67% Updates, 33% Lookups (c) 50% Updates, 50% Lookups

(d) 20% Updates, 80% Lookups (e) 10% Updates, 90% Lookups (f) 1% Updates, 99% Lookups

Figure 5: Scalability of the CA tree variants in various scenarios compared to `ordered_set` and `set` with concurrency options activated.

similar, but it remains as future work to find the exact root of this problem. It is not surprising that the CA tree variants scale worse than `ordered_set` in the read-only case since the CA trees also need to adapt to the contention. However, the CA trees still perform well and even beat the `set` implementation with fine-grained locking when several NUMA nodes are used.

In mixed operation scenarios (Figure 5) the CA trees outperform all other table types when several NUMA nodes are used. It is not surprising that the CA trees perform better than the `ordered_set` protected by a single readers-writer lock. On the other hand, it is less obvious why they scale better than `set` with fine-grained locking. One reason could be the `set`'s contended hot spots discussed earlier. Another point worth noting about the fine-grained locking in `set` is that its implementation currently contains a limited number of bucket locks (64 in Erlang/OTP 17.0) while the CA trees can adapt the number of locks that are used to the current contention level. A CA tree will also adapt to situations where the contention is distributed unevenly over the key range and create the fine-grained locks where needed.

5.2 Scalability with Different Set Sizes

Let us now analyze a benchmark that keeps the distribution of operations fixed (80% `lookup` operations and the rest updates) and varies the set size. We use key ranges of sizes between 2^3 to 2^{27}; cf. Figure 6. The insert phase that always happens before the mixed phase performs as many operations as half the size of the key range. We measure the time it takes for the worker processes to perform 2^{22} operations as in the previous benchmarks.

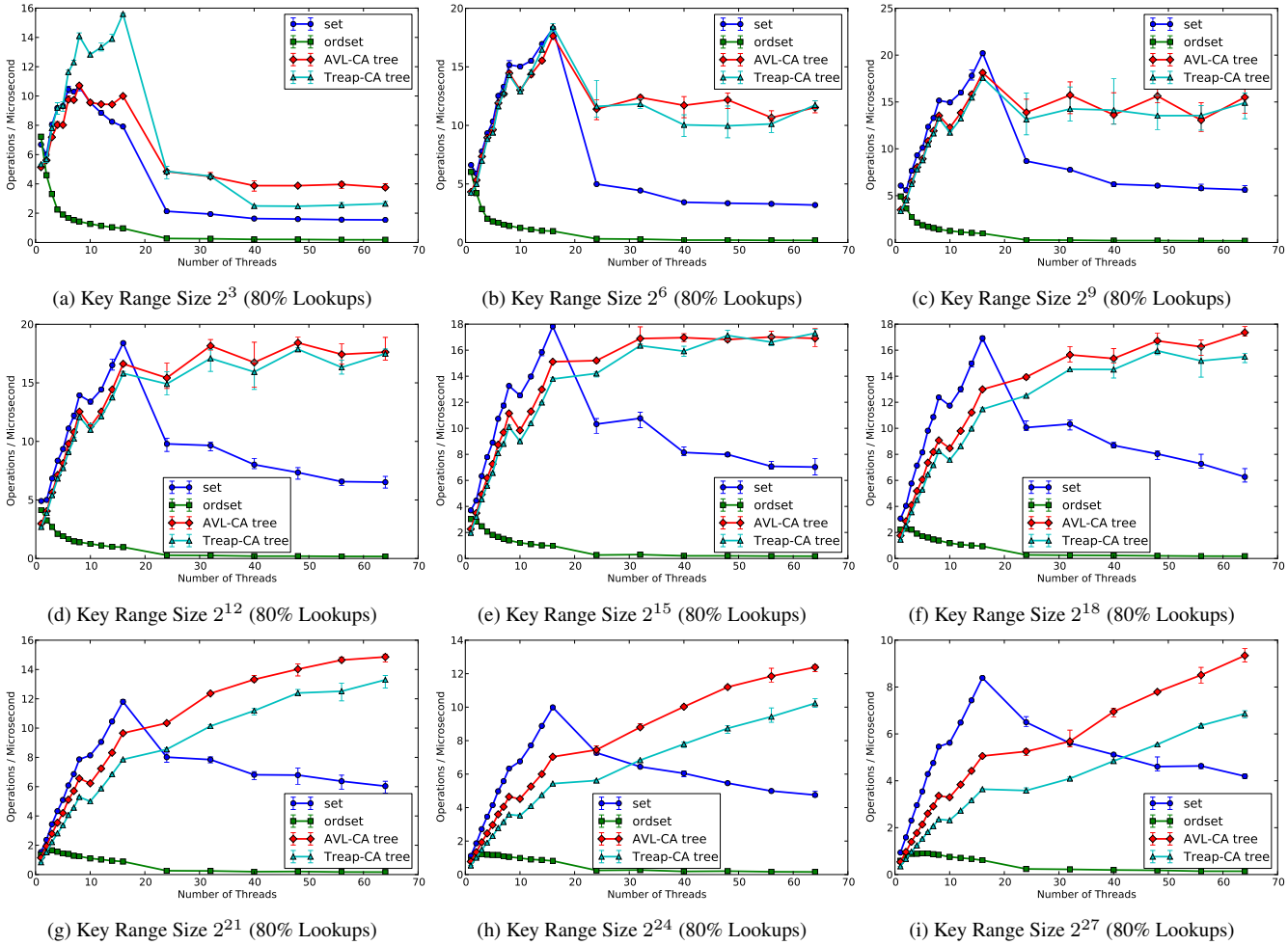
The number of locks in a CA tree is limited by the key range size. Therefore, it is not surprising that the CA trees seem to scale better with larger key range sizes. However, the CA trees scale surprisingly well even with a key range of eight (Figure 6a). With such a small key range, the locks will be contended and the performance will to a large degree depend on the lock implementation. In the current implementation of CA trees we use the mutex lock from the pthreads library. This is a sleeping lock as the locks used for the standard ETS table types and is thus friendly to other threads running in the system. A more aggressive lock implementation would probably improve the scalability of the CA trees when the key range is small.

The scalability of `ordered_set` improves by a small amount with larger key ranges. This is probably because the overhead of the lock becomes less visible when the operations take longer time. The overhead of the memory management hot spots in the CA trees are also more hidden when the operations take longer time. However, `set`'s scalability problem when more than one NUMA node is used, seems to persist even with the larger key ranges.

5.3 Sequential Performance

In Figure 7, the sequential performance measurements from the benchmark described in the previous section are displayed. The sequential performance difference between the AVL-CA tree and `ordered_set` is small for all set sizes. However, the relative difference between those two implementations is larger for small set sizes. Both `ordered_set` and the AVL-CA tree use the same AVL tree implementation, so the overhead of the AVL-CA tree just consists of the infrastructure needed to do the adaptation. One extra node (a base node) has to be traversed in the AVL-CA tree. There is also some overhead in maintaining a statistics counter and checking if adaption shall be made.

Another overhead that would be removed in a more mature implementation of the CA trees is the overhead induced by having two levels of table locks. The CA trees have been developed in a stand-alone data structure library to make testing and experimentation easier. The table lock in this library is used for safe memory reclamation as described in Section 4. In the current prototypical implementation both the table lock from ETS and from the stand-alone library are read-locked for normal operations. This makes integration into ETS very simple but also slightly inefficient since one of the table locks is redundant. The table locks in the stand-alone library use the same type of reader groups as the ones used in ETS. An integer variable in the thread's reader group is incremented with an atomic instruction during a read lock call and decremented in a read unlock call. These atomic instructions are relatively expensive since they also include a full memory barrier.



(a) Key Range Size 2^3 (80% Lookups)

(b) Key Range Size 2^6 (80% Lookups)

(c) Key Range Size 2^9 (80% Lookups)

(d) Key Range Size 2^{12} (80% Lookups)

(e) Key Range Size 2^{15} (80% Lookups)

(f) Key Range Size 2^{18} (80% Lookups)

(g) Key Range Size 2^{21} (80% Lookups)

(h) Key Range Size 2^{24} (80% Lookups)

(i) Key Range Size 2^{27} (80% Lookups)

Figure 6: Scalability of the CA tree variants with varying set sizes compared to `ordered_set` and `set` with concurrency options activated using 80% lookups and 20% updates.

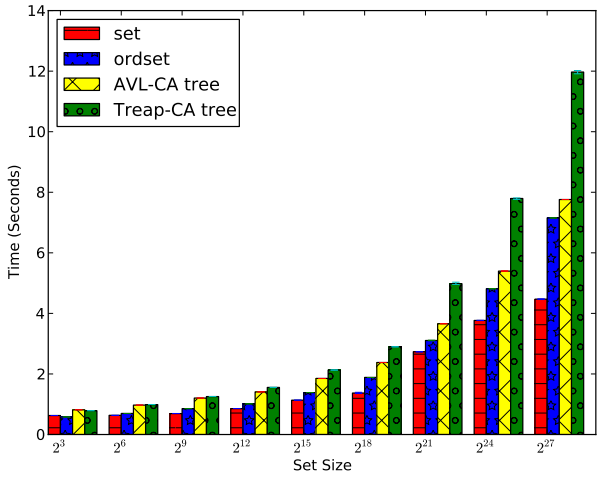


Figure 7: Sequential performance using 80% lookups and 20% updates and varying set sizes.

6. Related Work

The literature contains a vast number of concurrent ordered set data structures, e.g. [2, 6, 8, 9, 18]. It is beyond the scope of this paper to discuss all of them in detail, but we want to note that none of them is likely to have as low sequential overhead as the CA tree.

Some work specific to ETS has been published before. In particular, our own study about the scalability of ETS and its evolution across Erlang/OTP releases [12] is a predecessor to this paper. There we gave a detailed description of the ETS implementation and discussed some of its scalability problems.

Nyblom has suggested the addition of software transactional memory (STM) to ETS [19]. However, that STM implementation is concerned only with hash-based tables and not with `ordered_set`.

Fritchie has published a study about the ETS table implementations and their performance where he also compares the then current implementations of ETS with B-trees and Judy arrays [10]. However, Fritchie's study was concerned only with sequential performance and did not contain any discussion about concurrent execution.

In a recent work about more scalable libraries for queue delegation locking we experimented with the use of alternative locking libraries [13] for ETS. In particular, we compared the current readers-writer lock used as table lock in ETS with the DR-MCS [7] and the MR-QD lock [14]. We showed that DR-MCS and MR-QD

locks provide better performance for `ordered_set` when there are concurrent write operations. However, since readers-writer locks do not allow parallel write operations they can not give as good scalability as that provided by CA trees.

7. Discussion and Future Work

First of all, some work is still needed to make contention adapting trees ready for inclusion in the ETS code of the Erlang/OTP distribution. In particular, support for the full ETS API needs to be implemented, but, as explained in Section 4, we expect that this will be a relatively easy task.

If it is decided to integrate CA trees into ETS, one also has to decide in what way that should be done. One alternative is to replace the `ordered_set` implementation with the AVL-CA tree. This way the read-only scenarios will get slightly worse performance and scalability than presently. Another alternative would be to add an additional table type. The disadvantage of this approach is that it will complicate the ETS programming interface and add to the decision making process and possible experimentation and measurements that programmers have to do. Finally, a non intrusive option would be to only use the AVL-CA tree when `write_concurrency` is activated on an `ordered_set` table. This way the read-only cases can still get the current good performance and scalability while the scalability problems that `ordered_set` currently has in scenarios that contain write operations can be avoided.

The CA tree itself is an interesting concurrent set data structure since it is very simple to implement, has low sequential overhead and can adapt to contention. We intend to work more on contention adapting concurrent data structures; in particular to prove properties of CA trees and discuss their algorithms in detail. We will also try to improve the performance of CA trees by extending its basic implementation, for example by using readers-writer locks in the base nodes and by using hardware lock elision [20].

8. Concluding Remarks

We have proposed a new `ordered_set` implementation for ETS that can handle concurrent write operations. Our experiments show that the new implementation not only makes `ordered_set` more performant when there are concurrent write operations but also makes it scale well on big multicores in many scenarios. Furthermore, the new `ordered_set` implementation even scales better than ETS tables based on hashing with fine grained locking when several NUMA nodes are used. It can therefore help improve the performance and scalability of a wide range of Erlang applications.

Acknowledgments

This work has been supported in part by the European Union grant IST-2011-287510 “RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software” and by UPMARC (the Uppsala Programming for Multicore Architectures Research Center).

References

[1] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962. In Russian.

[2] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan. CBTree: A practical concurrent self-adjusting search tree. In *International Symposium on Distributed Computing*, pages 1–15. Springer, 2012. URL http://link.springer.com/chapter/10.1007/978-3-642-33651-5_1.

[3] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 540–545, Oct. 1989. . URL http://dx.doi.org/10.1109/SFCS.1989.63531.

[4] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma. Using read-copy-update techniques for system V IPC in the Linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309. USENIX, 2003. ISBN 1-931971-11-0. URL https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli.pdf.

[5] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A scalability benchmark suite for Erlang/OTP. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang*, pages 33–42. ACM, 2012. ISBN 978-1-4503-1575-3. URL http://doi.acm.org/10.1145/2364489.2364495.

[6] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–268. ACM, 2010. URL http://dx.doi.org/10.1145/1693453.1693488.

[7] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–166, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. URL http://doi.acm.org/10.1145/2442516.2442532.

[8] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *Euro-Par 2013 Parallel Processing - 9th International Conference*, volume 8097 of *LNCS*, pages 229–240. Springer, 2013. URL http://dx.doi.org/10.1007/978-3-642-40047-6_25.

[9] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.

[10] S. L. Fritchie. A study of Erlang ETS table implementations and performance. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, pages 43–55. ACM, 2003. URL http://dx.doi.org/10.1145/940880.940887.

[11] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007. ISSN 0743-7315. URL http://dx.doi.org/10.1016/j.jpdc.2007.04.010.

[12] D. Klaftenegger, K. Sagonas, and K. Winblad. On the scalability of the Erlang term storage. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, pages 15–26, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2385-7. URL http://doi.acm.org/10.1145/2505305.2505308.

[13] D. Klaftenegger, K. Sagonas, and K. Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par 2014, Proceedings of the 20th International Conference*, volume 8632 of *LNCS*. Springer, 2014. Preprint available from http://www.it.uu.se/research/group/languages/software/qd_lock_lib.

[14] D. Klaftenegger, K. Sagonas, and K. Winblad. Queue delegation locking, 2014. Available from http://www.it.uu.se/research/group/languages/software/qd_lock_lib.

[15] P.-Å. Larson. Linear hashing with partial expansions. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 224–232. VLDB Endowment, 1980.

[16] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[17] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15 (6):491–504, 2004. URL http://dx.doi.org/10.1109/TPDS.2004.8.

[18] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 317–328, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. URL http://doi.acm.org/10.1145/2555243.2555256.

[19] P. Nyblom. Erlang ETS tables and software transactional memory: How transactions make ETS tables more like ordinary actors. In *Proceedings of the Tenth ACM SIGPLAN Workshop on Erlang*, pages 2–13. ACM, 2011. URL http://dx.doi.org/10.1145/2034654.2034658.

[20] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1369-7. URL http://dl.acm.org/citation.cfm?id=563998.564036.

[21] K. Sagonas and K. Winblad. Technical report: Contention adapting trees, 2014. Available from http://www.it.uu.se/research/group/languages/software/ca_tree.