On the Scalability of the Erlang Term Storage

David Klaftenegger Konstantinos Sagonas Kjell Winblad

Department of Information Technology, Uppsala University, Sweden

firstname.lastname@it.uu.se

Abstract

The Erlang Term Storage (ETS) is an important component of the Erlang runtime system, especially when parallelism enters the picture, as it provides an area where processes can share data. It is therefore important that ETS's implementation is efficient, flexible, but also as scalable as possible. In this paper we document and describe the current implementation of ETS in detail, discuss the main data structures that support it, and present the main points of its evolution across Erlang/OTP releases. More importantly, we measure the scalability of its implementations, the effects of its tuning options, identify bottlenecks, and suggest changes and alternative designs that can improve both its performance and its scalability.

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

General Terms Experimentation, Measurement, Performance

Keywords concurrent data structures, multi-core, Erlang

1. Introduction

Multicore computers are becoming increasingly commonplace. Erlang programs can benefit from this, as the current Erlang runtime system comes with support for multiple schedulers (running in separate operating system threads) and is thus able to run Erlang processes in parallel. Unfortunately, this is not enough to guarantee scalability as the number of processor cores increases. Resource sharing and the associated synchronization overhead can easily become a scalability bottleneck on multicore machines.

The Erlang/OTP system provides two main ways of communicating between processes, namely message passing and shared Erlang Term Storage (ETS) tables. Message passing is often the most natural way to do interprocess communication. However, for some applications, inserting and updating data stored in a shared memory area, such as an ETS table, is a more appropriate alternative. For example, the parallel version of many algorithms maintains a "state" data structure which needs to be continuously accessible by all processes rather than being passed around either as a `spawn` argument or as part of a message. In such scenarios, even in Erlang, using a shared ETS table is often the most reasonable implementation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Erlang '13, September 28, 2013, Boston, MA, USA.
Copyright © 2013 ACM 978-1-4503-2385-7/13/09…$15.00.
http://dx.doi.org/10.1145/2505305.2505308

ETS is a central component of Erlang/OTP. It is used internally by many of its libraries and it is the underlying infrastructure of main memory databases such as `mnesia`. (In fact, every ETS table can be seen as a key-value store or an in-memory database table.) Furthermore, ETS is heavily used in Erlang applications. Out of 190 open source Erlang projects that we searched, at least 41 had a direct reference to a shared ETS table.[1] We see two major reasons why ETS is heavily used in Erlang projects. The first reason is convenience. ETS provides a standard way to have a key-value store. Having a standard way to do a common task makes it easier for programmers to understand new code and avoids duplication of the same functionality. The second reason is performance. A big portion of ETS is written in C, as part of the Erlang runtime system. It would be very difficult, if not impossible to implement ETS purely in Erlang with similar performance. Due to its reliance on mutable data, the functionality of ETS tables is very expensive to model in a functional programming language like Erlang.

To use an ETS table, one needs to first create one using the `new/2` function, which returns a reference to the new table. This reference can then be passed to all other ETS functions to specify on which table they should operate. For example, the `insert/2` function takes such a reference in its first argument, and either a tuple with a key-value pair to be stored or a list of such tuples in its second argument, and inserts them atomically in the corresponding ETS table. Other operations include looking up an entry with a specific key, finding all terms matching a specified pattern, deleting an entry or atomically deleting a list of entries with specific keys, etc. In addition, it is possible to iterate over a table and visit all its elements. A special function, called `safe_fixtable/2`, allows this to be done even while other operations work on the table. Iteration starts with a call to the `first/1` function and then calling `next/2` repeatedly, until the end of the table is reached. Note, that depending on the type of the table the order of elements may or may not be defined. More high level operations exists to, for example, query a table for all tuples matching a specified pattern.

All ETS tables have a type: either `set`, `bag`, `duplicate_bag` or `ordered_set`. The `set` type does not allow two elements in the table with the same key. The `bag` type allows more than one elements with the same key but not more than one element with the same value. The `duplicate_bag` type allows duplicate elements. Finally, the `ordered_set` type is semantically equivalent to the `set` type, but allows traversal of the elements stored in the order specified by the keys. In addition, it is possible to specify access rights on ETS tables. They can be `private` to an Erlang process, `protected` i.e. readable by all processes but writable only by their owner, or `public` which means readable and writable by all

[1] The open source projects selected were the ones with more than 10 watchers on `http://github.com` and projects mentioned on the Erlang mailing list. References in libraries used by the searched projects may be included, if the source code repository contains the library source. The search through the code is rather simplistic and only finds `ets:new/2` calls with options `public` or `protected` passed on the same line.

processes in an Erlang node. In this paper we focus on ETS tables which are shared between processes.

When processor cores write or read to shared ETS tables, they need to synchronize to avoid corrupting data or reading an inconsistent state. ETS provides an interface to shared memory which abstracts from the need of explicit synchronization, handling it internally. If a shared ETS table is accessed in parallel from many cores at the same time, the performance of the application can clearly be affected by how well the ETS implementation is able to handle parallel requests. Ideally, we would like the time per operation to be independent of the number of parallel processes accessing the ETS table. This goal is in practice not possible to achieve for operations that need to change the same parts of the table. (However, it might be possible to accomplish it when parallel processes access different parts of a table.) We measure the scalability of an ETS table as the amount of parallel operations that can be performed on the table without getting a considerable slowdown in time per operation.

More specifically, in this paper we:

1. document and describe the implementation and evolution of the Erlang Term Storage across Erlang/OTP releases, focussing on its support for parallel accesses;
2. evaluate these implementations and the various ETS tuning options in terms of scalability and performance; and
3. discuss improvements and possible alternative designs.

The rest of this paper is structured as follows. In the next section we review the details of the low-level implementation of ETS and in Section 3 we present the evolution of its support for parallelism. After briefly reviewing related work in Section 4, the main part of the paper measures the performance and scalability of ETS' implementation across different Erlang/OTP releases (Section 5) and presents some ideas on how to further improve its performance and scalability (Section 6). The paper ends with concluding remarks.

2. Low-Level Implementation

In this section, we describe in detail aspects of the current ETS implementation (that of Erlang/OTP R16B) that affect performance and scalability.

Memory Management The Erlang runtime system is currently based on processes that have process-local stacks and heaps (i.e., heaps whose cells cannot be referenced from any point other than from the process itself). The main benefit of this organization is that memory management of processes can take place at any time without requiring any sort of synchronization with other processes. Another invariant of the current Erlang runtime system is that the only terms that reside outside the process heaps are statically known constants, big binaries and bitstrings. Because processes are collected on an individual basis and can die at any point, data that they store into public ETS tables, presumably for other processes to access, currently needs to be copied there upon insertion. Likewise, when reading data from a shared ETS table this data is copied into the process-local heap, because the table owner can also terminate at any point. With this implementation scheme, ETS tables can be managed independently of their owning process, and memory management does not have to track users of tables for safe deallocation.

Backend Data Structures The current implementation of tables of type `ordered_set` is based on the AVL tree data structure [1]. The other three types (`set`, `bag` and `duplicate_bag`) are based on the same linear hash table implementation [10], their only difference being in how they handle duplicate keys and duplicate entries. In the benchmarks presented in later sections we have selected the `set` type as representative for all hash based table types.

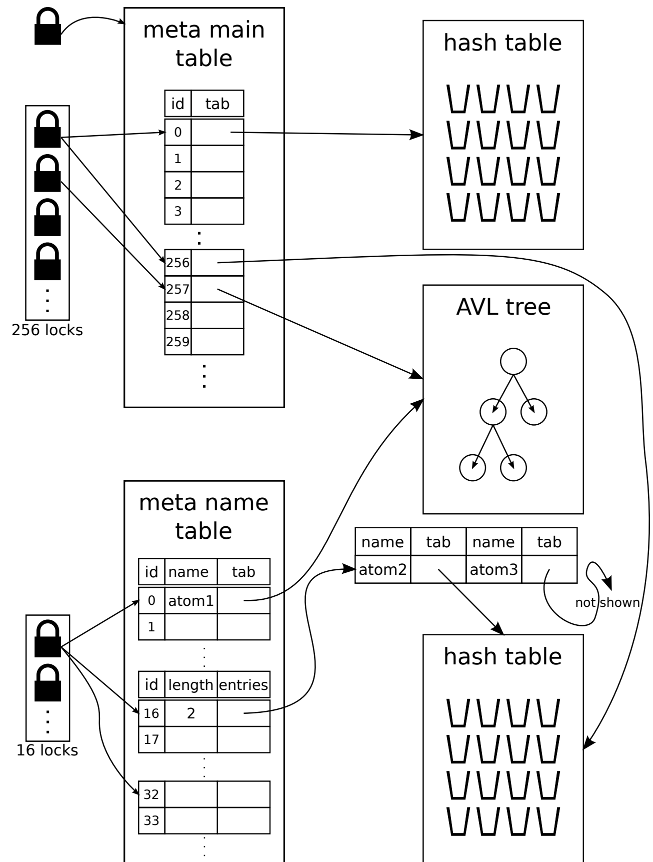

Figure 1. ETS meta tables: In this example, there are three ETS tables. An unnamed table with TID 0, and two named tables with TIDs 257 and 256. The tables with TIDs 0 and 256 use hashing, while the table with TID 257 is based on an AVL tree. TIDs 0 and 256 use the same lock in the `meta_main_table`. This is not a problem here, as named tables are accessed through names instead of TIDs. Tables `atom1` and `atom2` use the same lock in `meta_name_table`. This can be a bottleneck if both tables are often accessed at the same time. As `atom2` and `atom3` hash to the same position in the `meta_name_table`, an additional list of tables in this bucket is used for them.

Table Fixation When a table that is based on hashing is expanded or contracted, the keys in the table structure might be relocated. Fixating a table effectively disables growing and contracting and keeps information about deleted elements. Table fixation thus makes it possible to traverse a table, so every element that exists in the table at the beginning of the traversal will be returned at most once. Elements inserted during traversal may or may not be returned depending on their location in the internal table structure.

Per-Node Management Structures The following data structures are maintained on a node-wide level and are used for generic book keeping by the Erlang VM. Low-level operations, like finding the memory location of a particular ETS table or handling transfers of ownership, use only these data structures. There are two *meta tables*, the `meta_main_table` and the `meta_name_table`. They are depicted in Figure 1. Besides these, there are mappings from process identifiers (PIDs) to tables they own and from PIDs to tables that are fixated by them.

meta_main_table contains pointers to the main data structure of each table that exists in the VM at any point during run-

time. Table identifiers (TIDs) are used as indices into the meta_main_table. To protect accesses to slots of this table, each slot is associated with a reader-writer lock, stored in an array called meta_main_tab_locks. The size of this array is set to 256. Its locks are used to ensure that no access to an ETS table is happening while the ETS table is constructed or deallocated. Additionally the meta_main_table has one mutual exclusion lock, which is used to prevent several threads from adding or deleting elements from the meta_main_table at the same time. Synchronization of adding and removing elements from the meta_main_table is needed to ensure correctness in the presence of concurrent creations of new tables.

meta_name_table is the corresponding table to meta_main_table for named tables.

meta_pid_to_tab maps processes (PIDs) to the tables they own. This data structure is used when a process exits to handle transfers of table ownership or table deletion.

meta_pid_to_fixed_tab maps processes (PIDs) to tables on which they called safe_fixtable/2.

Table Locking ETS tables use readers-writer locks to protect accesses from reading data while this data is modified. This allows accesses that only read to execute in parallel, while modifications are serialized. Operations may also lock different sets of resources associated with a particular operation on an ETS table:

- Creation and deletion of a table require acquisition of the meta_main_table lock as well as the corresponding lock in the meta_main_tab_locks array.
- Creation, deletion and renaming of a named table also require acquisition of the meta_name_table lock and the corresponding lock in the meta_name_tab_rwlocks array.
- Lookup and update operations on a table's entries require the acquisition of the appropriate lock within the ETS table as well as acquisition of the corresponding read lock in the meta_main_tab_locks or meta_name_tab_rwlocks array. Without using any fine-tuning options, each table has just one readers-writer lock, used for all entries.

3. Improvements Between Erlang/OTP Releases

ETS support for parallelism has evolved over time. Here we describe the major changes across Erlang/OTP releases.

3.1 Support for Symmetric Multiprocessing (R11B)

Erlang/OTP got support for symmetric multiprocessing (SMP) in R11B. But not all runtime system components came with scalable implementations at that point. We define fine grained locking support in ETS tables as support for parallel updates of different parts of the table. In Erlang/OTP R11B, no ETS table type had any support for fine grained locking. Instead, each table was protected by a single reader-writer lock. As we will see, the scalability of the ETS implementation in R11B is not so good.

3.2 Support for Fine Grained Locking (R13B02-1)

Optional fine grained locking of tables implemented using hashing (i.e. tables of types set, bag and duplicate_bag) was introduced in Erlang/OTP R13B02-1. The fine grained locking could be enabled by adding the term {write_concurrency,true} to the list of ets:new/2 options. A table with fine grained locking enabled had one reader-writer lock for the whole table and an additional array containing 16 reader-writer locks for the buckets. The bucket locks are mapped to buckets in the way depicted in Figure 2. The mapping can be calculated efficiently by calculating bucket_index modulo lock_array_size.

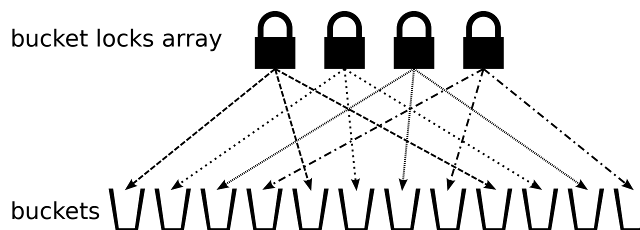

Figure 2. Mapping of locks to buckets using an array of four locks.

With the additional bucket locks protecting the buckets, a write operation can happen in parallel with other write and/or read operations. With write_concurrency enabled an ets:insert/2 operation that inserts a single tuple will:

1. acquire the right meta table lock for reading;
2. acquire the table lock for reading;
3. release the meta table lock;
4. find the bucket where the tuple should be placed;
5. acquire the corresponding bucket lock for writing; and finally
6. insert the tuple into the bucket before releasing the bucket lock and the read table lock.

Read operations need to acquire both the table lock and the bucket lock for reading when write_concurrency is enabled compared to just acquiring the table lock for reading when this option was not available. Thus enabling the write_concurrency option can make scenarios with just read operations slightly more expensive. Hence this option was not (and still is not) on by default.

Most operations that write more than one tuple in an atomic step, such as an ets:insert/2 operation inserting a list of tuples, acquire the table lock for writing, instead of taking all the needed bucket locks. (I.e., taking a single write lock was deemed more performing than taking many locks which would likely lock large parts of the table anyway.)

3.3 Reader Groups for Increased Read Concurrency (R14B)

All shared (i.e., public and protected) ETS tables have a table reader-writer lock. This is true even for tables with fine grained locking, since many operations need exclusive access to the whole table. However, since all read operations, and with fine grained locking even many common write operations (e.g. insert/2, insert_new/2, and delete/2) do not need exclusive access to the whole table, it is crucial that the reader part of the lock is scalable.

In a reader-writer lock, a read acquisition has to be visible to writers, so they can wait for the reads to finish before succeeding to take a write lock. One way to implement this is to have a shared counter that is incremented and decremented atomically when reading threads are entering and exiting their critical section. The shared counter approach works fine as long as the read critical section is long enough. However, it does not scale very well on modern multicore systems if the critical section is short, since the shared counter will entail a synchronization point. This synchronization cost is significant, even on modern processors which have a special instruction for incrementing a value atomically.

Figure 3 illustrates the problem of sharing a counter between several threads. The cache line holding the counter needs to be transferred between the two cores because of the way modern memory systems are constructed [15]. Transferring memory between private caches of cores is particularly expensive on Non-Uniform Memory Access (NUMA) systems, where cores can be located on

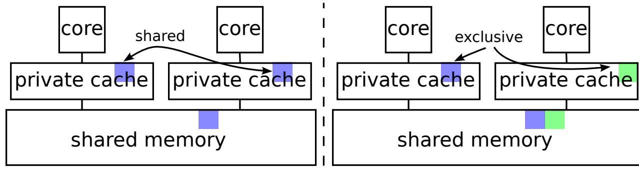

Figure 3. Illustration of cache line invalidation: When there is only a single counter (left, counter in blue), it can only be written from one core efficiently. It is invalidated in all other cores' caches on update. Using multiple counters in separate cache lines (right, counters in blue and green) avoids this problem. When used exclusively by one core, invalidation is unnecessary, so the counter stays cached.

different chips, connected only by a slower interconnect with limited bandwidth. The reader counter can instead be distributed over several memory words located in different cache lines. This makes writing slightly more expensive, since a writer needs to check all reader counters, but reading will scale much better.

Erlang/OTP R14B introduced the option `read_concurrency` that can be activated by specifying {`read_concurrency,true`} in the list of `ets:new/2` options when creating an ETS table. This option enables so called reader groups for reader counters in the ETS tables' reader-writer locks. A reader group is a group of schedulers (possibly just one in every group) that indicate reading by writing to a separate cache line for the group. Erlang scheduler threads are mapped to reader groups so that the threads are distributed equally over the reader groups. The default number of reader groups in Erlang/OTP R14B is 16, but can also be set as a runtime parameter to the Erlang VM. A scheduler indicates a read by incrementing a counter in the memory area allocated for its reading group.

3.4 More Refined Locking and More Reader Groups (R16B)

As mentioned in Section 3.2, the `write_concurrency` option enables fine grained locking for the hash based tables. Before R16B the buckets in the hash table were divided between 16 buckets. The size of the lock array was increased in Erlang/OTP R16B from 16 to 64 to give better scalability on machines with many cores. The default number of reader groups was also increased in Erlang/OTP R16B from 16, which was the previous default, to 64. We will evaluate the effect of these two changes in Sections 5.2.2 and 6.1.

4. Related Work

Here we briefly discuss other work which is relevant to ETS and its implementation. First, we review some past efforts to improve the performance of ETS. Second, we discuss data structures that could be used instead of or in addition to the current implementation, focusing on how they could improve its scalability.

4.1 Past Work on ETS

As ETS is a crucial component of the Erlang runtime, there have been previous attempts to improve its scalability and performance.

An ETS performance study comparing the current backends to a Judy array based implementation was presented by Fritchie [6]. A Judy array is a data structure similar to a trie, using compression to improve space and cache utilization. While Judy arrays are themselves ordered, they require keys to be converted from Erlang's internal memory format. The conversion does not preserve the Erlang term order, thus this implementation has to be considered as unordered. For large table sizes, lookups and insertions seemed to be faster with Judy arrays. However, term deletion and table traversal seemed to be slower with Judy arrays than with the then existing implementations of `set` and `ordered_set`. However, Fritchie's study was performed in 2003, before Erlang got support for SMP and thus did not measure scalability. Its results are therefore not fully comparable to the current ETS implementation, where data structures not only have to perform well, but also provide scalability in parallel scenarios.

In 2011, Nyblom suggested the addition of software transactional memory (STM) support for ETS tables [14]. Adding STM to Erlang was claimed to fit nicely with the actor programming model that Erlang uses, especially since there is no way to ensure atomicity of a sequence of ETS operations without serializing the access by using an Erlang process as a proxy or implementing some kind of locking mechanism. Serializing accesses to data will scale badly on a multicore system and locking is difficult to use and does not fit well with Erlang's programming model. Thus, Nyblom suggested ETS STM interfaces and implemented an experimental STM prototype on top of ETS. Benchmarks of this prototype showed that STM gives better scalability than serializing the accesses using a process as a proxy. However, Nyblom also noted that there are concerns about how the STM will scale on systems with more cores and that more experiments need to be performed.

4.2 Alternative ETS Backends

The data structures we will now look at provide different levels of progress guarantees. We adopt the definitions of Herlihy and Shavit [9]. A thread can be delayed (blocked) arbitrarily long in a *blocking* operation when some other thread is not executed. An operation is *lock-free* (and thus non-blocking), if at least one concurrent thread can complete the operation in a bounded number of steps. With a *wait-free* operation, all threads executing it will complete in a bounded number of steps.

While it is beyond the scope of this work to implement and test all data structures we review below, we briefly discuss why they are interesting candidates for improving ETS's scalability.

4.2.1 Unordered Data Structures

Currently, the unordered ETS tables are backed by a linear hash table implementation. While there is some support for scalability in this implementation, we identified the following problems that inherently limit scalability:

- The number of bucket locks is fixed and can not adapt to the table size or the number of concurrent accesses.
- Writers require exclusive access to a whole bucket, preventing concurrent reads in the same bucket.
- Table expansion is sequential and happens one bucket at a time, which limits scalability when ETS tables change size often.
- The use of locks makes it difficult to provide some progress guarantees for the data structure.

The following lock-free data structures are interesting for ETS, as they have no locking issues. This promises greater scalability than the lock based hash table currently used in ETS, as all but the third problem outlined above are related to locks.

Lock-free hash table Michael introduced a lock-free hash table based on lock-free linked lists [11]. Michael's table could be more scalable than ETS tables backed by linear hashing, but the algorithm does not support resizing. While this makes it unfit for use in ETS, it may be a good choice for fixed size hash tables, like the `meta_name_table`.

Lock-free split-ordered list Shavit and Shalev have proposed a lock-free extendable hash table [16]. The table entries are stored in a single lock-free linked list, which is sorted in *split order*. Split order is the order in which the entries will be split into

other buckets when resizing the table. The data structure has an extendable bucket array containing pointers to nodes in the entry list. When finding an element, the key is first hashed to a position in the bucket array and the pointer to the bucket it followed. If the data structure is integrated as an ETS table type, the use of `ets:safe_fixtable/2` will be unnecessary, like it is already with tables of type `ordered_set`.

4.2.2 Ordered Data Structures

We will also describe some ordered data structures, which could be used for all currently available ETS table types, even though only the `ordered_set` type requires ordering. We will compare them primarily to the current AVL tree backend. Its current implementation uses a single reader-writer lock to protect the table, obviously providing little scalability. To accommodate for more parallelism, it is thus necessary to look for alternatives.

Concurrent relaxed AVL tree The relaxed variant of the AVL tree presented by Bronson et al. [4] is somewhat similar to the current backend of `ordered_set`. It uses invisible readers, a technique from STM [17] that validates the reads using version numbers in the data blocks. Its fast atomic clone operation is interesting for atomic iteration over all elements in an ETS table without stopping all other users of the table.

Concurrent B-Tree B-trees are popular in disk based databases, as they use less disk block accesses than other data structures. Bender et al. noticed that the access time differences between the processor cache and main memory is similar to that between main memory and disk. Based on this, they developed both locking and lock-free variants of a B-tree [3]. As the same reasoning applies to ETS as an in-memory database system, it could prove beneficial to use a cache-optimized data structure.

Concurrent skip list Herlihy et al. have proposed a concurrent skip list with a wait-free lookup operation and locking insert and delete operations [8]. Modifying operations lock only the nodes that will be modified, possibly requiring retries. This skip list has been implemented as an experimental backend for ETS tables, which is compared with current backends in Section 6.2.1. This particular skip list was chosen as, according to to its creators, it is easier to implement and prove correct than Fraser's lock free skip list [5, 8], even though Fraser's skip list is completely lock-free, while supposedly providing similar scalability and performance.

5. Performance and Scalability Study

Having described the implementation of ETS and its evolution, in this section we measure its performance and scalability across different Erlang/OTP releases and quantify the effect that tuning options have on the performance of ETS using the benchmark environment that we describe below.

Benchmark Machine and Runtime Parameters All benchmarks were run on a machine with four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz), eight cores each (i.e., a total of 32 cores, each with hyperthreading, thus allowing the Erlang/OTP system to have up to 64 schedulers active at the same time). The machine ran Linux 3.2.0-4-amd64 SMP Debian 3.2.35-2 x86_64 and had 128GB of RAM. For all benchmarks, the command line option `+sbt tnnps` was set, except when not available in the Erlang/OTP release (i.e. it was not used in R11B-5 since it was introduced in a later release). This option requests a thread pinning policy that spreads the scheduler threads over hardware cores, but one NUMA node at a time. When cores can handle more than one thread in hardware, the work is spread evenly among all cores before additional schedulers are bound to cores that are already in use. So on our machine, benchmarks with up to eight schedulers were run on one NUMA node, benchmarks with nine to sixteen schedulers were run on two NUMA nodes, and so on. Schedulers 33 through 64 are mapped to hyperthreads on cores already in use by another scheduler. This policy tries to use the available computation power, while minimizing the communication cost between the schedulers. See `+sbt` in the Erlang manual for more detailed information about the option. For the compilation of Erlang/OTP releases, GCC version 4.7.2 (Debian 4.7.2-5) was used.

Benchmark Description For benchmarking we used bencherl, a benchmarking infrastructure for measuring scalability of Erlang applications [2]. The benchmark we used is ets_bench, a benchmark measuring distinct times for three phases: insertion, access and deletion. The benchmark is designed to measure only those actions and no auxiliary work, like generating random numbers, or distributing them among worker processes. It sets up an ETS table prior to measuring, using `write_concurrency` and `read_concurrency` when the options are available in the Erlang/OTP release, unless specified otherwise. All ETS operations use uniformly distributed random keys between 1 and $2,097,152$. To make use of the available processing resources, the benchmark starts one worker process per available scheduler, while the number of schedulers is varied by bencherl. First, the insertion phase evenly distributes $1,048,576$ keys to the workers, and measures the time it takes for them to insert these keys into the table. To reduce the cost of coping memory, the tuples that we insert into the table contains just one element (the key). Secondly, the access phase generates $16,777,216$ pairs of keys and operations, where an operation is either a lookup, an insert, or a delete. To create different scenarios of table usage, we varied the percentage of lookup vs. update operations in the access phase; namely we created workloads with 90%, 99% and 100% lookups. For the remaining 10% and 1% updates, the probability for inserts and deletes is always identical, so that the size of the data structure should stay roughly the same. The measured runtime is only the time taken for this second stage of the benchmark.

Information on the Figures In all graphs, the x-axis shows the number of runtime schedulers and the y-axis shows time in seconds. To ensure that the benchmark results are reliable, we run each benchmark three times. The data points in the graphs presented in the following sections is the average run time of these three runs. We also show the minimum and maximum run time for the three runs as a vertical line at each data point.

5.1 Performance and Scalability Across OTP Releases

First, we measure scalability of the Erlang/OTP releases with major changes in ETS (R11B-5, R13B02-1, R14B, and R16B), for accessing tables of type `set` and `ordered_set` using the three workloads mentioned above. The results are shown in Figures 4–9. Note that the x-axis shows the number of schedulers and the y-axis shows run time in seconds.

5.1.1 Workloads with 90% Lookups and 10% Updates

As can be seen in Figure 4, the scalability of ETS tables of type `set` improved significantly starting with release R14B. As described in Section 3, the main change in this release was the introduction of reader groups. However, the scalability difference between R13B02-1 and R14B is unlikely to be caused by reader groups alone. Another important change between these two releases is that the locks in the `meta_main_tab_locks` array was changed from using mutual exclusion locks to reader-writer locks. In fact the reader-writer locks used for the meta table are of the new type with reader groups enabled. Even though the meta table lock is just acquired for a very short time in the benchmark (to read the

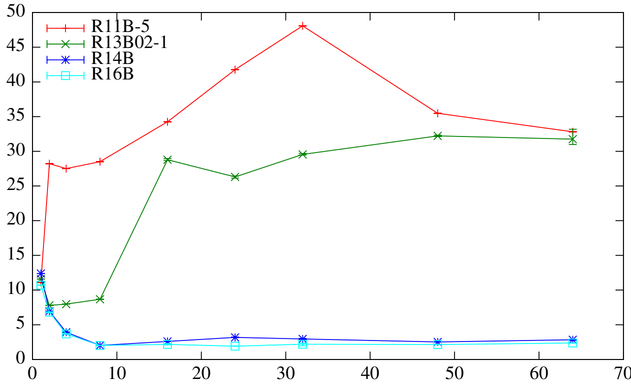

Figure 4. Scalability of ETS tables of type `set` across Erlang/OTP releases using a workload with 90% lookups and 10% updates.

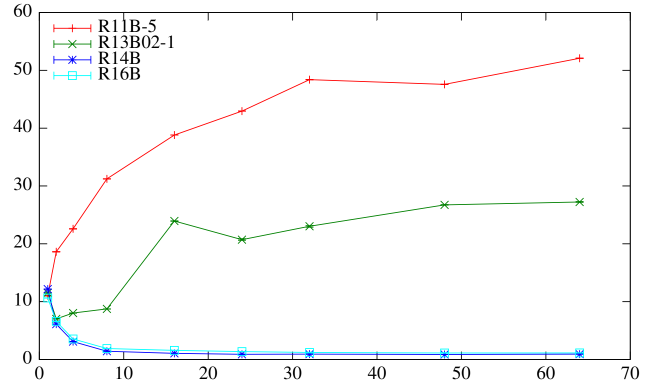

Figure 6. Scalability of ETS tables of type `set` across Erlang/OTP releases using a workload with 99% lookups and 1% updates.

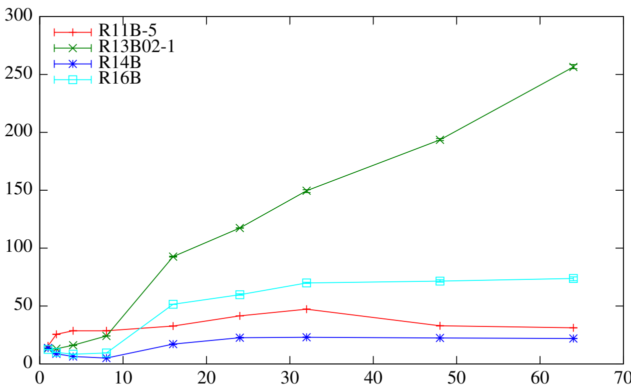

Figure 5. Scalability of ETS tables of type `ordered_set` across OTP releases using a workload with 90% lookups and 10% updates.

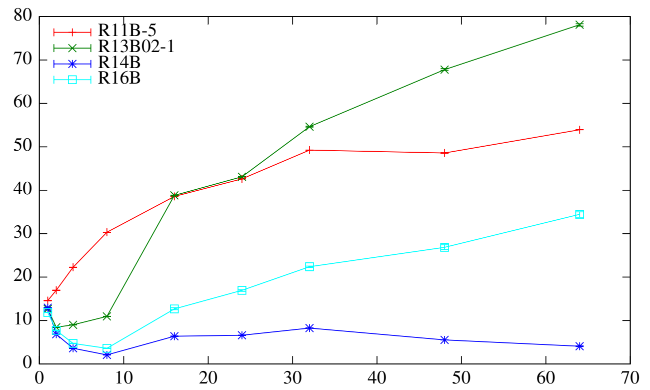

Figure 7. Scalability of ETS tables of type `ordered_set` across OTP releases using a workload with 99% lookups and 1% updates.

`meta_main_table`), it is reasonable to believe, after looking at Figure 4, that doing this is enough to give unsatisfactory scalability.

Figure 5 shows scalability of the same workload with a table of type `ordered_set`. An unexpected result here is that Erlang/OTP R14B performs better than R16B. In fact, R16B seems to scale even worse than R11B-5 on this workload. The main change, which affects tables of type `ordered_set`, between releases R14B and R16B is the number of reader groups. (The effect of reader groups on scalability is investigated further in Section 5.2.2.) For this workload, scalability is worst in R13B02-1. In R11B, the table's main reader-writer lock used was taken from pthreads [13], while in R13B02-1 a custom locking implementation was introduced for some platforms. More work needs to be done to find out which changes actually cause the difference in the results. The problem is that a lot of structural changes happened in R13B02-1 to support fine grained locking of ETS tables based on hashing, which makes it difficult to find the exact cause for the difference.

5.1.2 Workloads with 99% Lookups and 1% Updates

For an ETS table of type `set`, the results for this workload, shown in Figure 6, are very similar to those with 10% update operations. Compare them with those in Figure 4.

For a table of type `ordered_set`, whose implementation does not support fine grained locking, once again R14B achieves the best performance; see Figure 7. Also as can be seen from the same figure, the performance of the custom locking implementation in R13B02-1 is slightly better than that of R11B-5 for low numbers of schedulers. However, its performance significantly drops once the level of parallelism becomes high enough; especially once schedulers cannot be pinned in the same processor chip and effects from NUMA start to become noticeable. It is clear that the custom locking scheme was not designed for such levels of parallelism back in 2009. The good news is that its performance has improved since then, but on the other hand it is disturbing that the scalability of Erlang/OTP R16B gets so much worse than that of R14B.

5.1.3 Workloads with 100% Lookups

When measuring workloads with only lookups, the results and conclusions are the same for R11B-5 and R13B02-1; see Figures 8 and 9. As far as the performance of R16B vs. R14B is concerned, one can notice a scalability difference between ETS tables of type `set` vs. `ordered_set`. Intuitively, one would assume that, due to the absence of write locking, the scalability should be similar for both table types on this benchmark. Contrary to this expectation, the `set` table type scales better than `ordered_set`.

Upon closer examination we discovered that the AVL tree implementation internally uses a stack for walking through the tree. While this is perfectly normal, the problem comes from a memory allocation optimization, which statically allocates one such stack which is used whenever it is not already in use by another operation. When already in use, a dynamic stack is used instead. This results in a bottleneck: the static stack is transferred between schedulers. As the stack is written to, it cannot be cached locally, which increases the communication cost drastically. As explained, the pin-

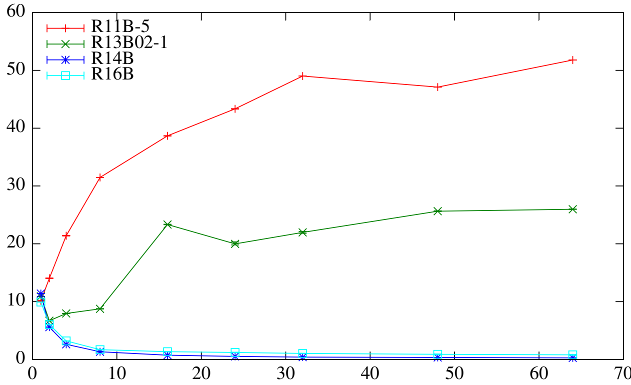

Figure 8. Scalability of ETS tables of type `set` across Erlang/OTP releases using a workload with 100% lookups.

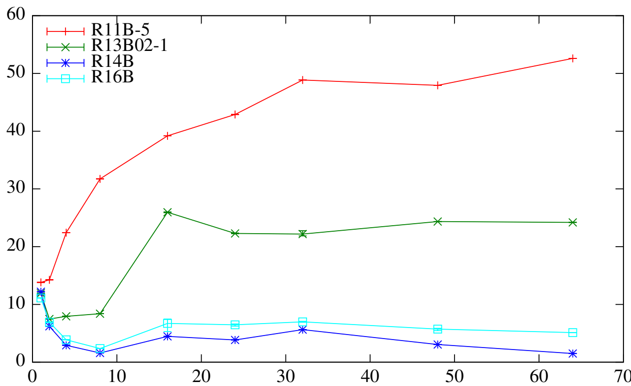

Figure 9. Scalability of ETS tables of type `ordered_set` across OTP releases using a workload with 100% lookups.

ning policy groups the first eight schedulers on the same processor chip, which gives it a lower latency for cache coherency messages. This is the reason for the static stack not having a negative impact for up to eight schedulers on this machine. The effect then gradually increases until the stack is almost continually used. After that, more parallelism results in more dynamic stacks being used, which scales better. The effect of this static stack is further investigated in Section 5.3.

5.2 Effect of Tuning Options

We have described the two ETS options `write_concurrency` and `read_concurrency` and the runtime command line option `+rg` in Section 3. Here we report on their effectiveness by measuring the performance effect for enabling vs. not enabling these options on the ets_bench benchmark run on Erlang/OTP R16B.

5.2.1 Effect of Concurrency Options

How options `write_concurrency` and `read_concurrency` influence the ETS data stuctures and locks is summarized in Table 1.

Figures 10, 11 and 12 show the performance results. The x-axis shows the number of schedulers and the y-axis shows run time in seconds. On the workloads with a mix of lookups and updates, it is clear that the only configurations that scale well are those with fine grained locking. Without fine grained locking, `read_concurrency` alone is not able to achieve any improvement on mixed workloads even when the percentage of updates is as low as 1%. In fact, for tables of type `ordered_set`, the option `read_concurrency` does

	no	r	w	r & w
set lock type	*normal*	*freq_r*	*freq_r*	*freq_r*
set bucket lock type	-	-	*normal*	*freq_r*
ordered_set lock type	*normal*	*freq_r*	*normal*	*freq_r*

Table 1. r denotes `read_concurrency` enabled, w denotes `write_concurrency` enabled, set lock type and ordered_set lock type refer to the type of the lock, *normal* means the default reader-writer lock without the reader groups optimization and *freq_r* means the reader-writer lock with reader groups optimization.

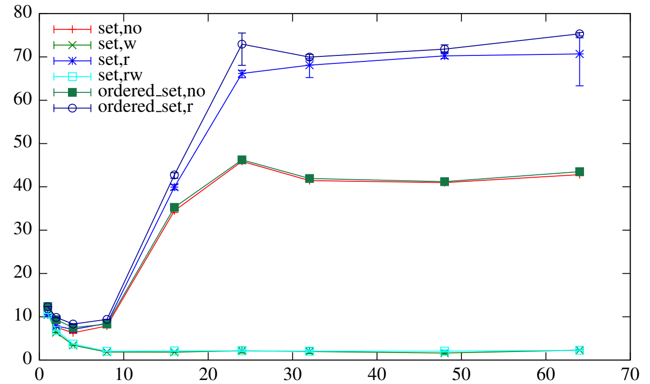

Figure 10. Scalability of ETS on a workload with 90% lookups and 10% updates when varying the ETS table concurrency options.

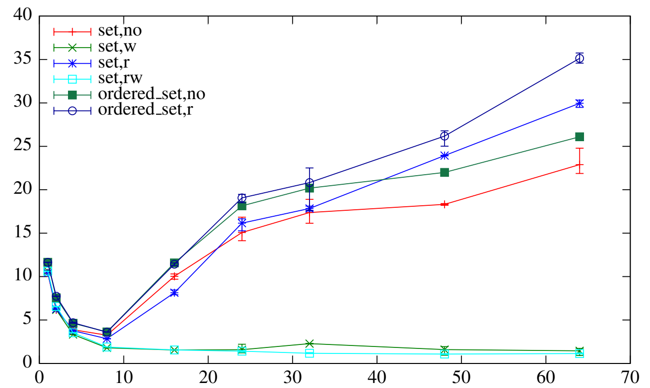

Figure 11. Scalability of ETS on a workload with 99% lookups and 1% updates when varying the ETS table concurrency options.

not manage to achieve any speedup even on lookup only workloads; cf. Figure 12. We will explain why this is so in Section 5.3.

Interestingly, for tables of type `set`, there is no significant gain in using both `read_concurrency` and `write_concurrency` compared to enabling just the `write_concurrency` option. On the other hand, the memory requirements are increased with a significant constant when both options are enabled compared to enabling just `write_concurrency`. With the default settings, a reader-writer lock with reader groups enabled requires 64 cache lines (given that at least 64 schedulers are used). Since there are 64 bucket locks in R16B and usually 64 bytes for a cache line that means at least $64 \times 64 \times 64$ bytes ($= 0.25$ MB) for every table with both `read_concurrency` and `write_concurrency` enabled.

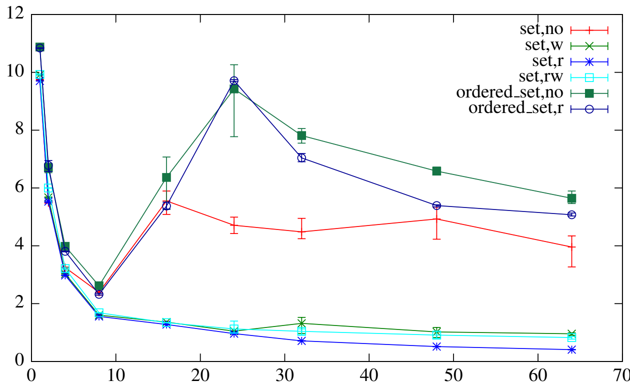

Figure 12. Scalability of ETS (`ordered_set`) on a workload with 100% lookups when varying the ETS table concurrency options.

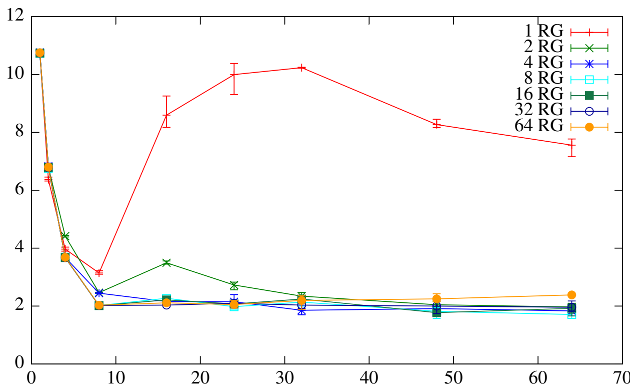

Figure 13. Scalability of ETS tables of type `set`, on 90% lookups and 10% updates, when varying the number of reader groups.

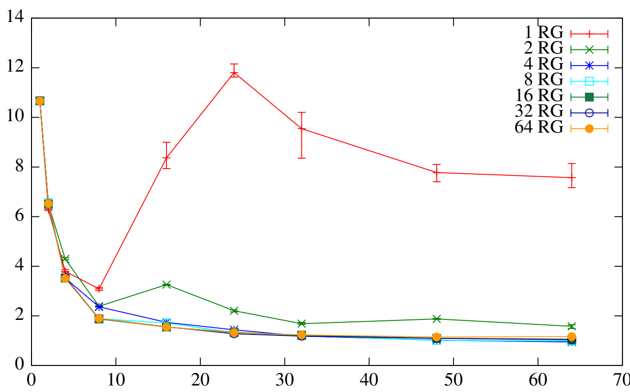

Figure 14. Scalability of ETS tables of type `set`, on 99% lookups and 1% updates, when varying the number of reader groups.

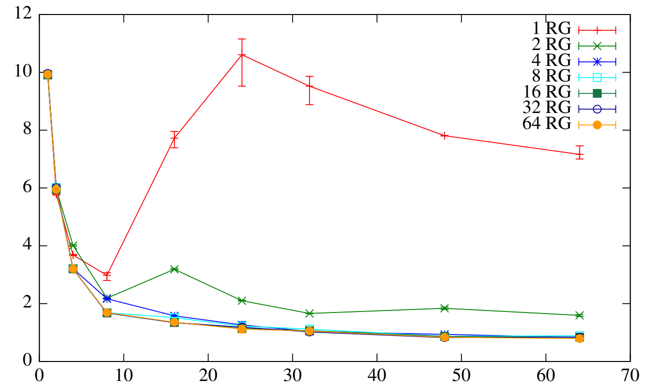

Figure 15. Scalability of ETS tables of type `set`, performing only lookups, when varying the number of reader groups.

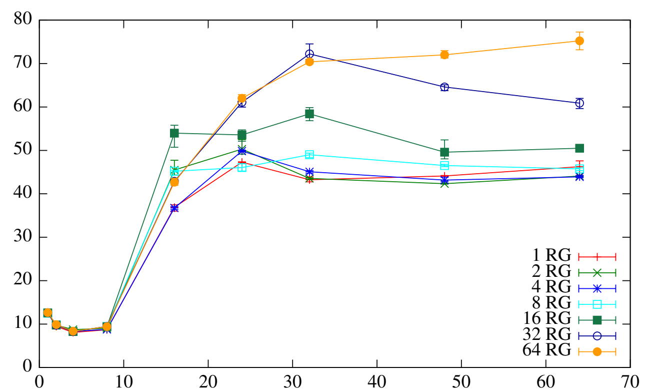

Figure 16. Scalability of tables of type `ordered_set`, on 90% lookups and 10% updates, varying the number of reader groups.

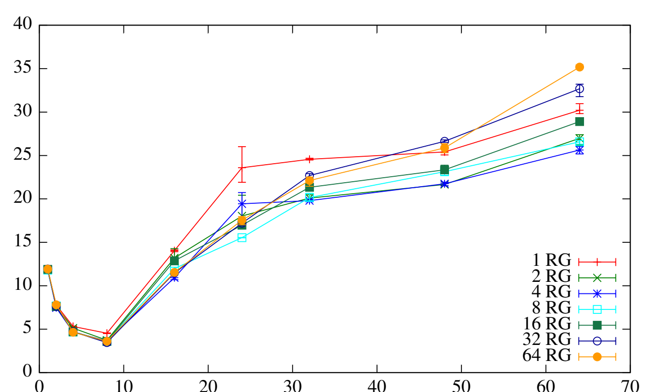

Figure 17. Scalability of tables of type `ordered_set`, on 99% lookups and 1% updates, varying the number of reader groups.

5.2.2 Effect of Reader Groups (`read_concurrency` and `+rg`)

To test the effect of the runtime system parameter `+rg`, which is used to set the maximum number of reader groups, we ran the `ets_bench` benchmark varying the number of reader groups. The default number of reader groups in R16B is 64. The actual number of reader group is set to the minimum of the number of schedulers and the value of the `+rg` parameter. The result of the benchmark is depicted in Figures 13–18.

It is clear from the figures showing the results for the `set` type (Figures 13, 14 and 15) that just having one reader group is not sufficient. For the workloads measured in the benchmark it seems like four or eight reader groups perform well. However, from 4 to 64 reader groups performance varies very little for tables of type `set`. It is worth noting that for `set`, the lock acquisitions are distributed over 64 bucket locks. Therefore, none of the bucket locks are likely to have many concurrent read and write accesses.

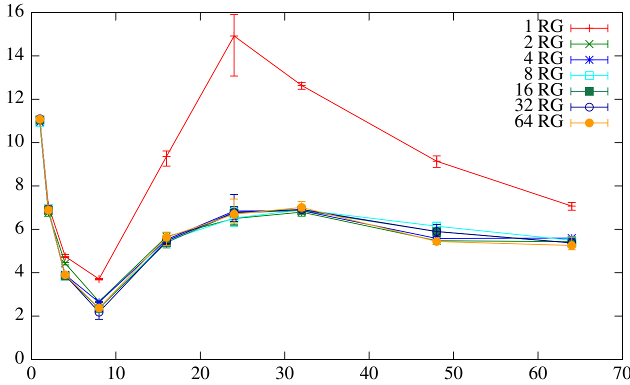

Figure 18. Scalability of tables of type `ordered_set`, on 100% lookups, when varying the number of reader groups.

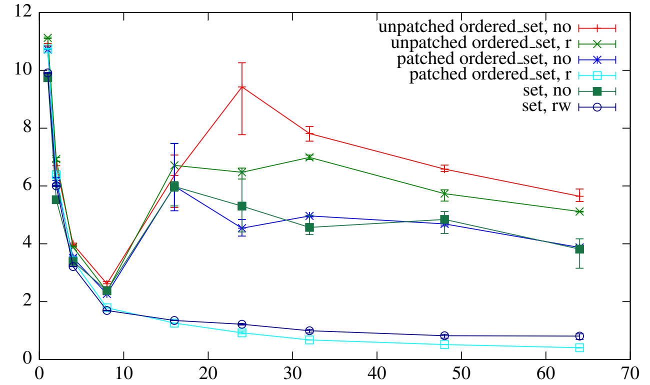

Figure 19. Scalability on a lookup-only workload when the AVL tree implementation uses a static (unpatched) vs. dynamic (patched) stack. (See Table 1 for the explanation of no, r and rw.)

The results for tables of type `ordered_set` are more intriguing. For these tables, it seems that many reader groups can hurt performance in mixed lookup and update workloads; see Figures 16 and 17. A good choice for the number of reader groups when using `ordered_set` in these cases seems to be two to eight. Two to eight reader groups also seems to perform comparatively well on the lookup only workload; cf. Figure 18.

It is worth noting that there are several reader-writer locks (that are affected by the `+rg` parameter) protecting different critical sections with different kinds of access patterns. For example, the reader-writer locks protecting the `meta_main_table` are expected to be very frequently acquired for reading, while the reader-writer locks protecting the buckets in `set` tables to be relatively frequently acquired for writing. Finding a setting that works for most use cases will probably be increasingly difficult as the number of cores per system grows. Theoretically, read only scenarios should benefit from as many reader groups as the number of schedulers, while write only scenarios should benefit from as few reader groups as possible.

Tables of type `ordered_set` do not support fine grained locking and thus the table's global lock quickly becomes a scalability bottleneck. Note that the lack of scalability in the absence of writers (cf. Figure 18) is due to an implementation detail of the AVL tree and not caused by the reader groups. See Section 5.3 for details.

5.3 A Bottleneck in the AVL Tree Implementation

Finally, we measure the effect of the static stack allocation in the AVL tree. (The use of this stack was mentioned in Section 5.1.3.) To measure its impact, we applied a patch to Erlang/OTP R16B, which causes all operations to use a dynamic stack instead.

When running `ets_bench` on this patched version of Erlang, the results for mixed operations did not change significantly. However, the results for the lookup-only workload, shown in Figure 19, changed drastically: `ordered_set`, based on the AVL tree, scales now even better than the hash table based `set`. This proves that this stack is indeed the bottleneck observed in Section 5.1.3. Removing it from the code path of lookups is therefore advisable.

6. Towards ETS with Better Scalability

In this section we present some ideas for extending or redesigning the ETS implementation in order to achieve even better scalability than that currently offered by Erlang/OTP R16B. These ideas range from allowing more programmer control over the number of bucket locks in hash-based tables, using non-blocking data structures or existing data structure libraries with better performance, to schemes for completely eliminating the locks in the meta table.

6.1 Number of Bucket Locks

As mentioned in Section 3.4 the number of bucket locks for the hash-based ETS tables was increased in Erlang/OTP R16B from 16 to 64. To understand how the number of bucket locks affects performance, `ets_bench` was run with varying number of bucket locks. Currently, the number of bucket locks can not be set by a runtime parameter, therefore a version of Erlang/OTP R16B was compiled for each number of bucket locks tested. The benchmark was run with `write_concurrency` and `read_concurrency` enabled.

Figures 20, 21 and 22 show the benchmark results for the three kinds of workloads we consider. A bigger number of bucket locks has, unsurprisingly, a positive impact on scalability up to a certain point where the positive effect wears off. The number of bucket locks for an ETS table should be decided as a trade off between different factors:

1. how many schedulers the Erlang VM starts with;
2. how often and by how many processes the table is accessed;
3. what type of access is common for the table (for example, extremely read heavy tables do not require as many bucket locks as extremely write intensive tables); and
4. the overhead of having more locks than required.

What is a good trade off depends on the application and might change as computer technology develops. So our recommendation is to add support for setting the number of bucket locks at runtime and per table in future Erlang/OTP releases.

6.2 Alternative Backend Data Structures

To ease development and experimentation with alternative schemes, we developed a minimalistic interface for hooking up data structure implementations that support the main ETS functionality. While this interface currently does not support all ETS operations and is not ready for production use, it allows us to plug in data structures with only minimal glue code. We implemented a few experimental backends for ETS to learn about its internals. Here we describe two of these implementations to showcase the kind of performance differences achievable by alternative backend data structures.

6.2.1 Concurrent Skip Lists as Alternative to AVL Trees

As a first experiment, we have implemented a variant of a concurrent skip list with wait-free lookups [8] and integrated it into ETS as an alternative table type that we call `cskiplist`. We selected this data structure because of its non-blocking property and because it provides efficient support for the main ETS operations

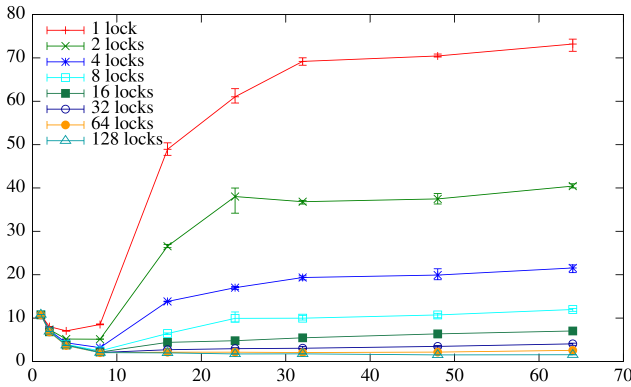

Figure 20. Effect of the number of bucket locks on a workload with 90% lookups and 10% updates on a table of type `set`.

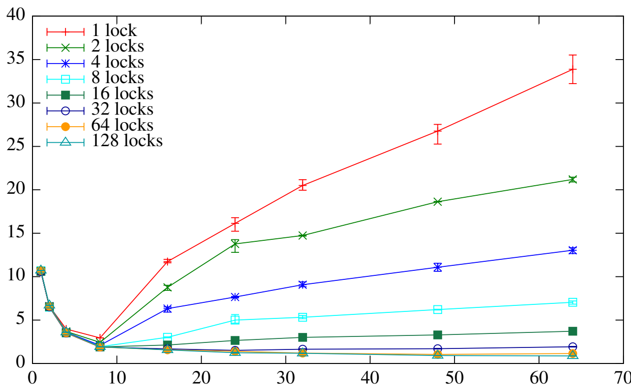

Figure 21. Effect of the number of bucket locks on a workload with 99% lookups and 1% updates on a table of type `set`.

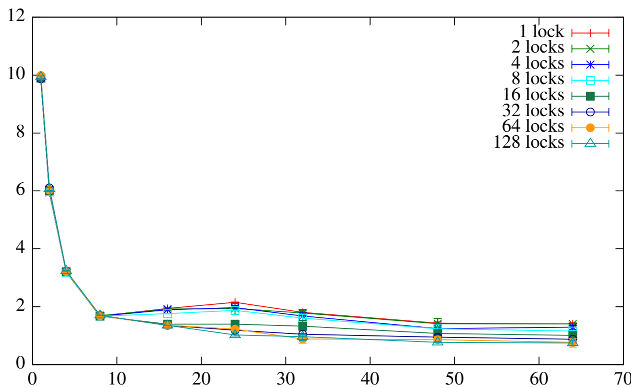

Figure 22. Effect of the number of bucket locks on a workload with only lookups on a table of type `set`.

(i.e. `insert`, `lookup` and `delete`) which was easy to implement. If needed, the remaining ETS operations can be trivially supported by acquiring a write lock on the whole table, but of course such an implementation cannot be expected to achieve good scalability.

In the `cskiplist` implementation, as well as in most other non-blocking data structures, memory blocks can be deleted while other threads are reading them. The memory management component needs to deal with this so that readers can not read memory that

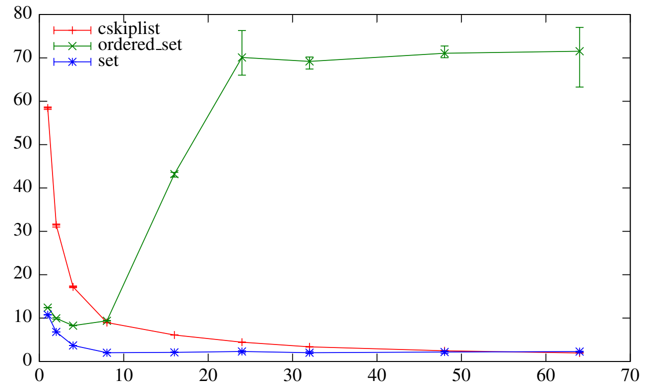

Figure 23. Comparison of a concurrent skip list against `set` and `ordered_set` on a workload with 90% lookups and 10% updates.

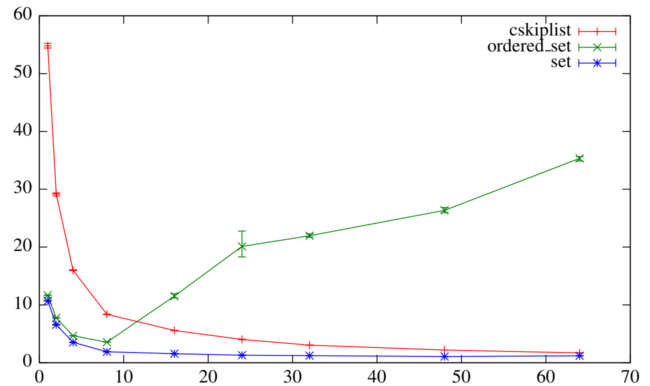

Figure 24. Comparison of a concurrent skip list against `set` and `ordered_set` on a workload with 99% lookups and 1% updates.

is not part of the data structure anymore. One solution is to use automatic memory management (garbage collection) and not free any memory blocks explicitly. Another solution, which is the one we selected for `cskiplist`, is provided by *hazard pointers* [12]. Hazard pointers require that every thread maintains:

- a thread local list of pointers to data that the thread currently accesses and can not be freed by other threads (*read list*) and
- a thread local list of pointers to data that will be freed later (*free list*).

When a thread detects that its free list has reached a certain size it scans the read list of all other threads and frees the memory blocks that are not found in these lists.

ETS tables implemented by `cskiplists` have been compared against `ordered_set` tables implemented by AVL trees and `set` tables implemented using linear hashing on the workloads we consider. The results of these comparisons are shown in Figures 23, 24 and 25. The `cskiplist` is very slow in the sequential case. It has been shown by Hart et al. [7] that memory reclamation strategies have a significant impact on the performance of data structures, and could therefore be a good starting point for improving this back-end implementation. However, `cskiplist` achieves much better scalability than the current `ordered_set` implementation for the workloads with both lookups and updates.

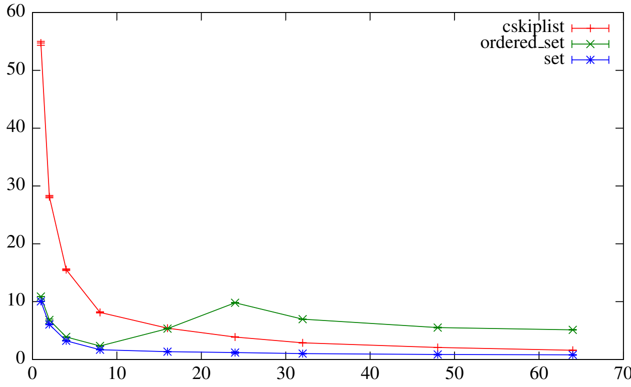

Figure 25. Comparison of a concurrent skip list against `set` and `ordered_set` on a workload with only lookups.

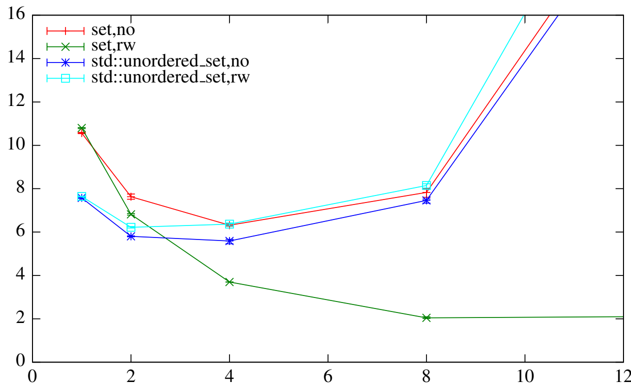

Figure 26. Comparison of the ETS hash table implementation against a hash table from the C++ standard library for a low number of schedulers on a workload with 90% lookups and 10% updates.

6.2.2 Using Data Structures from Available Libraries

The interface we developed has an additional layer which allows us to use C++ in addition to C. This both eases implementation of data structures and gives us access to a large range of data structure libraries. As a proof of concept, we wrote the necessary glue code to use `std::unordered_set` as a backend to ETS. It is a minimalistic hash table interface that comes with the standard library of C++11, though it does not allow all ETS functionality to be used. In particular, iteration over the table is not supportable and there is no fine grained locking support either. This means an ETS table based on `std::unordered_set` has to be locked completely on update operations, similar to the current implementation of tables of type `ordered_set`.

Figure 26 shows the performance of ETS tables using the `std::unordered_set` library compared to the current `set` implementation for low levels of parallelism. At high parallelism, the C++ implementation obviously cannot compete with `set` with `write_concurrency` due to requiring full locking for updates. One can observe however that the sequential execution is significantly faster for the `std::unordered_set` based table. This is likely caused by not having to support all the operations that the `set` implementation provides, and using an exponentially growing hash table instead of a linear one. Still, it raises the question whether ETS can profit from using faster data structures in cases where these features are not required by the application.

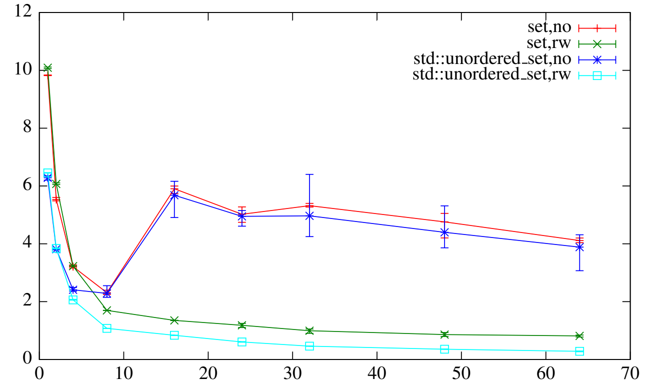

Figure 27. Comparison of current hash based ETS vs. a hash table from the C++ standard library on a workload with only lookups.

Especially in read-only workloads, as that of Figure 27, the benefits are clear. When not using `read_concurrency`, both backends perform similarly and are clearly limited by the performance of the lock used. Enabling `read_concurrency` exposes the bare performance difference between the two backends. The C++ library is 36% faster initially, and even 60% faster on 64 schedulers.

What is clear from this experiment that using feature-rich data structures comes at a price. This price may be acceptable for most use cases, but having the option to not pay it could be beneficial for some use cases. The performance of this experimental ETS table backend can be seen as a rough estimate how much performance one could gain from exposing more choices to the user, i.e. allow them to disable certain parts of the ETS interface that are costly in performance even when not used.

6.3 More Scalable Alternatives to Meta Table Locking

As described in Section 2, the ETS meta table is an internal data structure that maps table identifiers to the corresponding table data structures. The elements in the meta table are protected by readers-writer locks that we call meta table locks. The writer part of a meta table lock is only taken when tables are created or deleted. However, a meta table lock is acquired for reading every time a process executes an ETS operation. This lock might be a scalability problem since, if many processes access a single table frequently, there can be contention on memory bandwidth due to the write accesses to take the read lock.

In a prototype implementation of the Erlang VM, we have completely removed the need for meta table locks. Instead, the meta table is read and modified by atomic instructions. This approach leaves only ETS deletion as a problem, as tables can be deleted while readers access the data structure. To solve this issue, we have added a scheduler local pointer to the ETS table that is currently being accessed. The scheduler local ETS pointer is from here on simply called *ETS pointer*. Before a table is deallocated it is first marked as dead in the meta table and then the thread blocks until no *ETS pointer* is pointing to the ETS table.

In another prototype, an alternative locking scheme for the table lock based on the *ETS pointer*s was tested. In this locking scheme a read is indicated by setting the *ETS pointer* for the scheduler. This approach to read indicating is similar to the lock reader groups implementation described in Section 3.3 but with the advantage of using less memory. One disadvantage of the *ETS pointer* approach, compared to reader groups, is that in some scenarios it might be more expensive for writes, since *ETS pointer*s might be modified because of reads in other tables which might cause an additional cache miss for the writer. The cache miss happens because a write

instruction issued by one core invalidates the corresponding cache line in the private cache of all other cores. However, the extra memory that is required by the reader groups approach might also cause more cache misses if the cache is too small to fit all memory.

7. Concluding Remarks

We presented a brief overview of the internals of ETS and how its support for multicore machines has evolved over time. In addition, we measured the scalability of ETS, both across different Erlang/OTP releases and when using the various concurrency options and runtime system parameters that can influence its performance. From our scalability study we have identified a couple of problems with the current implementations and proposed improvements:

- The implementation of tables of type `ordered_set` does not scale well under mixed workloads. In fact, the current implementation has a scalability problem even on read-only scenarios which can be easily solved with a simple patch we described.
- Non-blocking data structures, such as the concurrent skip list we tried, would scale much better than the current implementation of `ordered_set`, but whether they can be made to be as fast as it for low levels of concurrency is still an open question.
- On the positive side, ETS tables based on hashing scale quite well when the `write_concurrency` option is enabled, at least on scenarios where the operations use randomly distributed keys. Still, lock-free data structures may be able to further improve scalability, especially when the choice of keys is biased heavily. It remains as future work to integrate and evaluate some lock-free data structures in ETS.

Regarding the various table concurrency options and tunable runtime parameters (e.g. number of reader groups, number of bucket locks, etc.), we believe that they are inherently difficult to get right because a good set up depends on both the hardware and the usage scenario. Furthermore, some tables might need more bucket locks to function well than other tables in the same system. Investigating scalable synchronization primitives and data structures that scale well on many different usage scenarios and that do not require extensive manual fine-tuning also remains as future work.

Acknowledgments

This work has been supported by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software" and by the UPMARC (Uppsala Programming for Multicore Architectures) research center. We thank Stavros Aronis for his involvement in the initial investigation of the ETS implementation.

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] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. A scalability benchmark suite for Erlang/OTP. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang*, pages 33–42. ACM, 2012.

[3] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 228–237. ACM, 2005.

[4] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle 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.

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

[6] Scott Lystig 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.

[7] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[8] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Structural Information and Communication Complexity*, pages 124–138. Springer, 2007.

[9] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.

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

[11] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82. ACM, 2002.

[12] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[13] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[14] Patrik 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.

[15] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 348–354. ACM, 1984.

[16] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006. See also expanded version available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.4226&rep=rep1&type=pdf` (accessed 9 June 2013).

[17] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.