RELEASE

SEVENTH FRAMEWORK PROGRAMME

ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

D2.4 (WP2): Robust Scalable Erlang Virtual Machine

Due date of deliverable: January 31, 2015
Actual submission date: March 17, 2015

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: Uppsala University Revision: 0.1 (March 17, 2015)

Purpose: To describe additions and improvements made to key components of the Erlang Virtual Machine that improve its robustness, scalability and responsiveness on big multicore machines.

Results: The main results presented in this deliverable are:

- Two new additions to the Erlang Virtual Machine (VM), namely a scheduler utilization balancing mechanism and the ability to interrupt long running garbage collecting BIFs, that improve the responsiveness of the Erlang/OTP system.
- Changes to the memory carrier migration mechanism, a new super carrier memory allocation scheme, and the new scheme for time management in the VM.
- A detailed description of the implementation of the Erlang Term Storage (ETS) and a study of the scalability and performance improvements to ETS across Erlang/OTP releases.
- New designs for ETS' implementation that improve its performance and scalability further.
- Scalable tracing support for profiling and monitoring SD Erlang applications.
- Scalability measurements of the VM improvements in Erlang/OTP releases during the project.

Conclusion: Additions and changes made during the RELEASE project have allowed many key components of the Erlang runtime system to become more efficient and scalable and have eliminated bottlenecks that previously hindered the scalability and responsiveness of its VM. Further improvements are possible by incorporating into the Erlang/OTP distribution components that currently exist in development branches of the system or in prototypes.

Project funded under the European Community Framework 7 Programme (2011-14)			
Dissemination Level			
PU	Public		✳
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Robust Scalable Erlang Virtual Machine

Konstantinos Sagonas <kostis@it.uu.se>
Sverker Eriksson <sverker.eriksson@ericsson.com>
Rickard Green <rickard.s.green@ericsson.com>
David Klaftenegger <david.klaftenegger@it.uu.se>
Kenneth Lundin <kenneth.lundin@ericsson.com>
Nikolaos Papaspyrou <nickie@softlab.ntua.gr>
Katerina Roukounaki <arou@softlab.ntua.gr>
Kjell Winblad <kjell.winblad@it.uu.se>

Contents

Executive Summary

This document is the fourth and last deliverable of Work Package 2 (WP2) of the RELEASE project. WP2 is concerned with improving the Erlang Virtual Machine (VM) by re-examining its runtime system architecture, identifying possible bottlenecks that affect its performance, scalability and responsiveness, designing and implementing improvements and changes to its components and, whenever improvements without major changes are not possible, proposing alternative mechanisms able to eliminate these bottlenecks. Towards this goal, we have gradually improved the implementation of several key components of the Erlang runtime system and complemented them with some new components offering additional functionality. Many of these have been presented already in Deliverables D2.2 and D2.3 and we will not repeat them here. In this document we describe additional components of the Erlang VM, additions and improvements to those described in D2.2 and D2.3, a deeper investigation of the performance and scalability characteristics of the Erlang Term Storage mechanism, and alternative data structures that further improve ETS' scalability.

More specifically, in this report:

- We present two new additions to the Erlang VM that improve the responsiveness of the Erlang/OTP system, namely a scheduler utilization balancing mechanism and the ability to interrupt long running garbage collecting built-in functions. (These additions are already part of the 17.x releases of Erlang/OTP.)
- We describe changes to the memory carrier migration mechanism presented in Deliverable 2.3, a new super carrier memory allocation scheme, and the new scheme for time management in the VM. (All these are scheduled to be included in the next major release of the system.)
- We perform a detailed study of the performance and scalability of ETS and the impact that support for fine-grained locking, reader groups and the concurrency options had to ETS' implementation.
- We propose alternative designs, based on contention adapting trees and queue delegation locks, for improving the performance and scalability of the ETS implementation even further.
- We document the support for profiling and monitoring added to the probes of the VM that allow it to profile SD Erlang applications.
- Finally, we include a selected set of scalability measurements showing the performance and scalability impact that the various changes and new components of the Erlang VM described in this and in previous deliverables of WP2 had to Erlang/OTP releases during the duration of the RELEASE project.

1 Introduction

The main goal of the RELEASE project is to investigate extensions of the Erlang language and improve aspects of its implementation technology in order to increase the performance of Erlang applications and allow them to achieve better scalability when run on big multicores or clusters of multicore machines. Work Package 2 (WP2) of RELEASE aims to improve the Erlang VM. The lead site of WP2 is Uppsala University. The tasks of WP2 pertaining to this deliverable are:

Task 2.2: "*... investigate alternative implementations of the Erlang Term Storage mechanism ...*"

Task 2.3: "*... investigate alternative runtime system architectures that reduce the need for copying data sent as messages between processes and scheduler extensions that reduce inter-process communication and support fine-grained parallelism.*"

Task 2.4: "*... design and implement lightweight infrastructure for profiling applications while these applications are running and for maintaining performance information about them.*"

Towards fulfilling these tasks, this fourth and last deliverable of WP2, presents new VM components, additions and changes to existing ones described previously in Deliverables D2.2 and D2.3, alternative designs for some others, and measures the scalability and performance benefits that all these have brought to the Erlang VM and the various versions of the Erlang/OTP system released during the duration of the project.

The remaining five sections describe these additions and changes in detail and, whenever appropriate, present scalability and performance measurements that show their impact to the system. Specifically, these sections describe:

- Two new additions to the Erlang Virtual Machine (VM), namely a scheduler utilization balancing mechanism and the ability to interrupt long running garbage collecting built-in functions, that improve the responsiveness of the Erlang/OTP system. Also, changes to the memory carrier migration mechanism, a new super carrier memory allocation scheme, and the new scheme for time management in the VM.
- A detailed description of the implementation of ETS, a study of the scalability and performance improvements to ETS across Erlang/OTP releases, and alternative designs, based on contention adapting trees and queue delegation locks, for improving the performance and scalability of the ETS implementation even further.
- Tracing support in VM probes for scalable profiling and monitoring of SD Erlang applications.
- The changes to BenchErl that were done to support the benchmarking and profiling of the Sim-Diasca application.
- A selected set of scalability measurements of the VM improvements in Erlang/OTP releases during the project.

The document ends with a brief section with some concluding remarks. The appendix lists changes and scalability improvements which have made it into the Erlang/OTP system in one of its releases during the duration of the RELEASE project.

The work for this deliverable has been done by researchers from Ericsson AB (EAB), the Institute of Communication and Computer Systems (ICCS), and Uppsala University (UU). The breakdown was the following:

- the new components and changes to the Erlang VM described in Section 2 of this document were done by the EAB team;
- the scalable tracing support for profiling and monitoring (Section 4) and the BenchErl changes to support benchmarking and profiling of Sim-Diasca (Section 5) were done by the ICCS team;
- the scalability investigation, performance improvements, and alternative schemes for the implementation of the Erlang Term Storage (Section 3) and the scalability benchmarking of Erlang/OTP releases (Section 6) were performed by the UU team.

The rest of this document contains a detailed description of the above.

2 Scalable Virtual Machine Architecture

Throughout the RELEASE project, the Erlang Virtual Machine (VM) and its RunTime System (ERTS) have been extended with a number of components that have either been developed from scratch or re-designed to improve the scalability, reliability and responsiveness of the Erlang/OTP system[1]. Deliverable D2.3 has presented most of them (Thread Progress, Delayed Deallocation, Carrier Migration, Process and Port Tables, Process Management, Port Signals, Code Loading,

[1]The appendix contains a lengthy list of these components and the command-line options that control them.

Trace Setting, and Term Sharing) in detail; we refer the interested reader to that document for their description.

During the last year of RELEASE, there have been improvements and additions to these components. For example, the algorithm of Carrier Migration has been redesigned, and some other VM features (Scheduler Utilization Balancing, Interruptible Garbage Collection) are already part of the Erlang/OTP distribution. Two more ERTS components (Super Carrier and Time Management) have also been implemented and are scheduled to be part of the next major release (18.0) of the Erlang/OTP system. This section will present the most important of these.

2.1 Scheduler Utilization Balancing

A new optional *scheduler utilization balancing* mechanism has been introduced in Erlang/OTP 17.0. It can be enabled by the programmer via the `+sub` command-line option. By default scheduler utilization balancing of load is disabled; instead scheduler compaction of load is enabled which will strive for a load distribution which causes as many scheduler threads as possible to be fully loaded (i.e., not run out of work). When scheduler utilization balancing is enabled (`+sub true`) the VM will instead try to balance scheduler utilization between schedulers. That is, it will strive for equal scheduler utilization on all schedulers.

Characteristic impact The scheduler utilization balancing mechanism has no performance impact on the system when not enabled. When enabled, it results in changed timing in the system; normally there is a small overhead due to measuring of utilization and calculating balancing information. On some platforms, such as old Windows systems, the overhead can be quite substantial. This time measurement overhead highly depends on the underlying primitives provided by the operating system.

2.2 Interruptible Garbage Collection

A call to either the `garbage_collect/1` or the `check_process_code/2` built-in function (BIF) may trigger garbage collection of a different process than the process calling the BIF. The previous implementations performed these kinds of garbage collections without considering the internal state of the process being garbage collected. In order to be able to more easily and more efficiently implement yielding native code, these types of garbage collections have been rewritten. A garbage collection like this is now triggered by an asynchronous request signal, the actual garbage collection is performed by the process being garbage collected itself, and finalized by a reply signal to the process issuing the request. Using this approach processes can disable garbage collection and yield without having to set up the heap in a state that can be garbage collected.

Additions and changes The `garbage_collect/2` and `check_process_code/3` BIFs have been introduced. Both take an option list as last argument. Using these, one can issue asynchronous requests. The `code:purge/1` and `code:soft_purge/1` BIFs have been rewritten to utilize asynchronous `check_process_code` requests in order to parallelize work.

Characteristic impact A call to the `garbage_collect/1` or the `check_process_code/2` BIF will normally take longer time to complete. On the positive side, the system as a whole is not as much negatively effected by the operation as before. A call to `code:purge/1` and `code:soft_purge/1` may complete faster or slower depending on the state of the system while the system as a whole is not as much negatively effected by the operation as before.

2.3 Changes to Carrier Migration

As described in Deliverable D2.3, the ERTS memory allocators manage memory blocks in single-block and multi-block chunks of raw memory, called raw memory *carriers*. A carrier is typically

created using `mmap()` on Unix systems. An allocator instance typically manages a mixture of single- and multi-block carriers.

In order to migrate carriers between allocator instances we move them through a *carrier pool*. In order for a carrier migration to complete, one scheduler needs to move the carrier into the pool, and another scheduler needs to take the carrier out of the pool. The pool is implemented as a lock-free, circular, doubly-linked list. The list contains a sentinel that is used as the starting point when inserting to, or fetching from the pool. Carriers in the pool are elements in this list. How migration of carriers between scheduler specific allocator instances of the same allocator type takes place has been described in Deliverable D2.3. But this pool also needs to be searched for free blocks.

2.3.1 Searching the Pool

To preserve real-time characteristics, searching the pool is limited. We only inspect a limited number of carriers. If none of those carriers has a free block large enough to satisfy the allocation request, the search fails. A carrier in the pool can also be busy, if another thread is currently doing block deallocation work on the carrier. A busy carrier will be skipped by the search as it can not satisfy the request. As mentioned, the pool is lock-free and we do not want to block waiting for the other thread to finish.

Before Erlang/OTP 17.4 When an allocator instance needs more carrier space, it always begins by inspecting its own carriers that are waiting for thread progress before they can be deallocated. If no such carrier is found, the allocator instance inspects the pool. If no carrier can be fetched from the pool either, the allocator instance will allocate a new carrier. Regardless of where the allocator instance gets the carrier from, it just links the carrier into its data structure of free blocks.

After Erlang/OTP 17.4 The old search algorithm had a problem as the search always started at the same position in the pool, the sentinel. This could lead to contention from concurrent searching processes. But even worse, it could lead to a "bad" state when searches fail with a high rate leading to new carriers instead being allocated. These new carriers may later be inserted into the pool due to bad utilization. If the frequency of insertions into the pool is higher than successful fetching from the pool, memory will eventually get exhausted.

This "bad" state consists of a cluster of small and/or highly fragmented carriers located at the sentinel in the pool. The largest free block in such a "bad" carrier is rather small, making it not able to satisfy most allocation requests. As the search always started at the sentinel, any such "bad" carriers that had been left in the pool would eventually form a bad cluster at the sentinel. Thus, all searches first had to skip past this cluster of "bad" carriers to reach a "good" carrier. When the cluster got to the same size as the search limit, the searches would essentially fail.

To counter the "bad cluster" problem and also ease the contention, the search now always starts by first looking at the allocator's **own** carriers. That is, it starts from carriers that were initially created by the allocator itself and later had been abandoned to the pool. If none of the abandoned carriers would do, then the search continues into the pool, as before, to look for carriers created by other allocators. However, if there is at least one abandoned carrier of the allocator that could not satisfy the request, we can use that as entry point into the pool.

The result is that we prefer carriers created by the thread itself, which is good for performance in NUMA machines. And we get more entry points when searching the pool, which will ease contention and clustering.

To do the first search among own carriers, every allocator instance has two new lists: `pooled_list` and `traitor_list`. These lists are only accessed by the allocator itself and only contain the allocator's own carriers. When an owned carrier is abandoned and put in the pool, it is also linked into `pooled_list`. When we search our `pooled_list` and find a carrier that is no longer in the pool, we move that carrier from `pooled_list` to `traitor_list` as it is now employed by another allocator. If searching `pooled_list` fails, we also do a limited search of `traitor_list`. When finding an

abandoned carrier in `traitor_list` it is either employed, or moved back to `pooled_list` if it could not satisfy the allocation request.

When searching `pooled_list` and `traitor_list` we always start at the point where the last search ended. This to avoid clustering problems and increase the probability to find a "good" carrier. As `pooled_list` and `traitor_list` are only accessed by the owning allocator instance, they need no thread synchronization at all.

Furthermore, the search for own carriers that are scheduled for deallocation is now done as the last search option. The idea is that it is better to reuse a poorly utilized carrier, than to resurrect an empty carrier that was just about to be released back to the operating system.

2.3.2 Result and Further Work

The use of this strategy of abandoning carriers with poor utilization and reusing them in allocator instances with an increased carrier demand is extremely effective and completely eliminates the problems that otherwise sometimes occurred when CPU load dropped while memory load did not.

When using the `aoffcaobf` or `aoff` strategies compared to `gf` or `bf`, we lose some performance since we get more modifications in the data structure of free blocks. This performance penalty is however reduced using the `aoffcbf` strategy. A trade-off between memory consumption and performance is however inevitable, and it is up to the user to decide what is most important.

As further work it would be quite easy to extend this to allow migration of multi-block carriers between all allocator types. More or less the only obstacle is maintenance of the statistics information.

2.4 Super Carrier

A *super carrier* is large memory area, allocated at VM start, which can be used during runtime to allocate normal carriers from.

The super carrier feature was introduced in OTP R16B03. It is enabled with command line option `+MMscs <size in MB>` and can be configured with other options.

2.4.1 Problems

The initial motivation for this feature was Erlang/OTP customers asking for a way to pre-allocate physical memory at VM start for it to use.

Other problems were some limitations experienced in the implementation of `mmap` in different operating systems:

- Increasingly bad performance of `mmap`/`munmap` as the number of mmap'ed areas grows.
- Fragmentation problem between mmap'ed areas.

Finally, a third problem was management of low memory in the half-word emulator. The implementation used a naive linear search structure to hold free segments which would lead to poor performance when fragmentation increased.

2.4.2 Solution

Allocate one large continuous area of address space at VM start and then use that area to satisfy our dynamic memory need during runtime. In other words: implement our own `mmap`.

Use cases If command line option `+MMscrpm` (Reserve Physical Memory) is set to false, only virtual space is allocated for the super carrier from start. The super carrier then acts as an "alternative mmap" implementation without changing the consumption of physical memory pages. Physical pages will be reserved on demand when an allocation is done from the super carrier and be unreserved when the memory is released back to the super carrier.

If `+MMscrpm` is set to true, which is default, the initial allocation will reserve physical memory for the entire super carrier. This can be used by users who want to ensure a certain *minimum* amount of physical memory for the VM.

However, what reservation of physical memory actually means highly depends on the operating system, and how it is configured. For example, different memory overcommit settings on Linux drastically change the behaviour.

A third feature is to have the super carrier limit the *maximum* amount of memory used by the VM. If `+MMsco` (Super Carrier Only) is set to true, which is default, allocations will only be done from the super carrier. When the super carrier gets full, the VM will fail due to out of memory. If `+MMsco` is false, allocations will use `mmap` directly if the super carrier is full.

Implementation The entire super carrier implementation is kept in `erl_mmap.c`. The name suggests that it can be viewed as Erlang's own `mmap` implementation.

A super carrier needs to satisfy two slightly different kinds of allocation requests: *multi block carriers* (MBC) and *single block carriers* (SBC). They are both rather large blocks of continuous memory, but MBCs and SBCs have different demands on alignment and size.

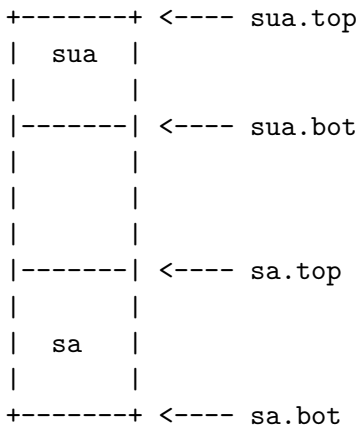
SBCs can have arbitrary size and only need minimum 8-byte alignment.

MBCs are more restricted. They can only have a number of fixed sizes that are powers of 2. The start address needs to have a very large alignment (currently 256 KB, called *super alignment*). This is a design choice that allows very low overhead per allocated block in the MBC.

To reduce fragmentation within the super carrier, it is good to keep SBCs and MBCs apart. MBCs with their uniform alignment and sizes can be packed very efficiently together. SBCs without demand for alignment can also be allocated quite efficiently together. But mixing them can lead to a lot of wasted memory when we need to create large holes of padding to the next alignment limit.

The super carrier thus contains two areas. One area for MBCs growing from the bottom and up. And one area for SBCs growing from the top and down. Like a process with a heap and a stack growing towards each other.

Data structures The MBC area is called `sa` (as in super aligned) and the SBC area is called `sua` (as in super un-aligned). Note that the "super" in super alignment and the "super" in super carrier have nothing to do with each other. Perhaps it would have been better if we had chosen different name prefixes to avoid confusion, such as "meta" carrier or "giant" alignment.



When a carrier is deallocated a free memory segment will be created inside the corresponding area, unless the carrier was at the very top (in `sa`) or bottom (in `sua`) in which case the area will just shrink down or up.

We need to keep track of all the free segments in order to reuse them for new carrier allocations. One initial idea was to use the same mechanism that is used to keep track of free blocks within MBCs (`alloc_util` and the different strategies). However, that would not be as straight forward as one can think and can also waste quite a lot of memory as it uses prepended block headers.

The granularity of the super carrier is one memory page (usually 4KB). We want to allocate and free entire pages and we do not want to waste an entire page just to hold the block header of the following pages.

Instead we store the meta information about all the free segments in a dedicated area apart from the `sa` and `sua` areas. Every free segment is represented by a descriptor struct (`ErtsFreeSegDesc`).

```
typedef struct {
    RBTNode snode;      /* node in 'stree' */
    RBTNode anode;      /* node in 'atree' */
    char* start;
    char* end;
} ErtsFreeSegDesc;
```

To find the smallest free segment that will satisfy a carrier allocation (best fit), the free segments are organized in a tree sorted by size (`stree`). We search in this tree at allocation. If no free segment of sufficient size is found, the area (`sa` or `sua`) is instead expanded. If two or more free segments with equal size exist, the one at lowest address is chosen for `sa` and highest address for `sua`.

At carrier deallocation, we want to coalesce with any adjacent free segments, to form one large free segment. To do that, all free segments are also organized in a tree sorted in address order (`atree`).

So, in total we keep four trees of free descriptors for the super carrier; two for `sa` and two for `sua`. They all use the same red-black-tree implementation that supports the different sorting orders used.

When allocating a new MBC we first search after a free segment in `sa`, then try to raise `sa.top`, and then as a fallback try to search after a free segment in `sua`. When an MBC is allocated in `sua`, a larger segment is allocated which is then trimmed to obtain the right alignment. Allocation search for an SBC is done in reverse order. When an SBC is allocated in `sa`, the size is aligned up to super aligned size.

The free descriptor area As mentioned above, the descriptors for the free segments are allocated in a separate area. This area has a constant configurable size (`+MMscrfsd`) that defaults to 65536 descriptors. This should be more than enough in most cases. If the descriptors' area fills up, new descriptor areas will be allocated first directly from the OS, then from `sua` and `sa` in the super carrier, and lastly from the memory segment itself which is being deallocated. Allocating free descriptor areas from the super carrier is only a last resort, and should be avoided, as it creates fragmentation.

Half-word emulator The half-word emulator uses the super carrier implementation to manage its low memory mappings that are needed for all term storage. The super carrier can here not be configured by command line options. One could imagine a second configurable instance of the super carrier used by high memory allocation, but that has not been implemented.

2.5 Time Management

Time management in the virtual machine has recently become a major scalability bottleneck. This as other more severe bottlenecks have been removed.

The main API for obtaining time information is `erlang:now/0`. It returns time since Epoch with micro second resolution. We call this time Erlang system time. `erlang:now/0` guarantees that values returned are strictly increasing. This time is the basis for all time internally in the virtual machine.

The Erlang system time should preferably align with the operating system's view of time since Epoch, i.e., OS system time. The OS system time can however be changed both forwards and backwards without limitation. The Erlang system time cannot be changed that way due to the guarantee

of returning strictly increasing values. The Erlang system time is therefore slowly adjusting towards OS system time if they do not align.

One problem with the time adjustment made is that we on purpose present time with a faulty frequency. Another problem is that the Erlang system time and OS system time can differ for very long periods of time. In order to solve this, we now introduce a solution similar to what is used on most operating systems. That is, a monotonic time that has its zero point at some unspecified point in time. Monotonic time is not allowed to make leaps forwards and backwards while system time is allowed to do this. The Erlang system time is just an offset from Erlang monotonic time.

2.5.1 Time Retrieval

Retrieval of the Erlang system time was previously protected by a global mutex which made the operation thread safe, but scaled poorly.

The Erlang system time and Erlang monotonic time need to run at the same frequency, otherwise the time offset between them would not be constant.

In the common case, monotonic time delivered by the operating system is solely based on a machine local clock while the system time is NTP adjusted. That is, they will run with different frequencies. Linux is an exception. It has a monotonic clock that is NTP adjusted and runs with the same frequency as system time.

In order for Erlang monotonic time to run with the same frequency as the Erlang system time, we need to adjust the frequency of the Erlang monotonic clock. This is done by comparing monotonic time and system time delivered by the OS, and calculate an adjustment.

In order to be able to do this in a scalable way, one thread calculates the time adjustment to use. If the adjustment needs to be changed, new adjustment information is published. This is done at least once a minute.

When a thread needs to retrieve time, it reads the monotonic time delivered by the OS and the time adjustment information previously published and calculates Erlang monotonic time.

It is important that all threads that read the same OS monotonic time map this to exactly the same Erlang monotonic time, otherwise Erlang monotonic time would not be monotonic. In order for this to be possible we need to synchronize the updates of the adjustment information. This is done by use of a readers-writer (RW) lock. The RW lock is write-locked only when the adjustment information is changed. This means that in the vast majority of cases the RW lock will be read-locked, which allows multiple readers to run concurrently. In order to prevent cache-line bouncing of the lock cache-line we use our own reader optimized RW lock implementation where reader threads notify about their presence in separate cache-lines.

2.5.2 Timer Wheel

The timer wheel mechanism, which was described in detail in Deliverable 2.3, contains all timers set by Erlang processes. The previous implementation was protected by a global mutex. This solution of course scaled poorly. In order to alleviate this, each scheduler thread is assigned its own timer wheel that is used by processes executing on the scheduler.

2.5.3 BIF Timer

The BIF timer implementation also was protected by a global mutex. Besides inserting timers into the timer wheel the BIF timer implementation also has to take care of a mapping between a reference that identifies the BIF timer and the actual timer in the timer wheel.

In order to get a scalable BIF timer solution we have implemented scheduler specific BIF timer servers as Erlang processes. The BIF timer servers keep information about timers in private ETS tables and only insert one timer at the time into the timer wheel.

2.5.4 Benchmarks

We have run two micro benchmarks on a dual 8-core AMD Opteron 4376 HE machine; i.e, the machine has a total of 16 physical cores.

The first benchmark tests the cost of inserting timers in the timer wheel by sending messages to processes that have a `receive after` clause and compares this with sending messages to processes that do a `receive` without an `after` clause. When there is an `after` clause in a `receive` statement a timer has to be set when the process blocks in the `receive`, and canceled when a message is put into the message queue. When executing this benchmark using Erlang/OTP 17.4, the total execution time when using timers is 62% longer than without timers. When running this benchmark on a system with the above optimizations, the total execution time when using timers is only 5% longer than without timers.

The second benchmark repeatedly checks the Erlang system time. When running on Erlang/OTP 17.4 this is done by calling `erlang:now/0`. When running on the optimized system this is done by calling the new API functions `erlang:monotonic_time/0` and `erlang:time_offset/0` and adding the result. The speedup when using the optimized system compared to Erlang/OTP 17.4 is more than 6900%.

3 Scalable Erlang Term Storage

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 used internally by many of the Erlang/OTP 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.)

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 processes in an Erlang node. In this document we focus on ETS tables that are shared between processes (i.e., `public` or `protected`).

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.

3.1 Low-Level Implementation

The current implementation of ETS 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 [9], their only difference being in how they handle duplicate keys and duplicate entries. In the benchmarks presented in Section 3.3 we have selected the `set` type as representative for all hash-based table types.

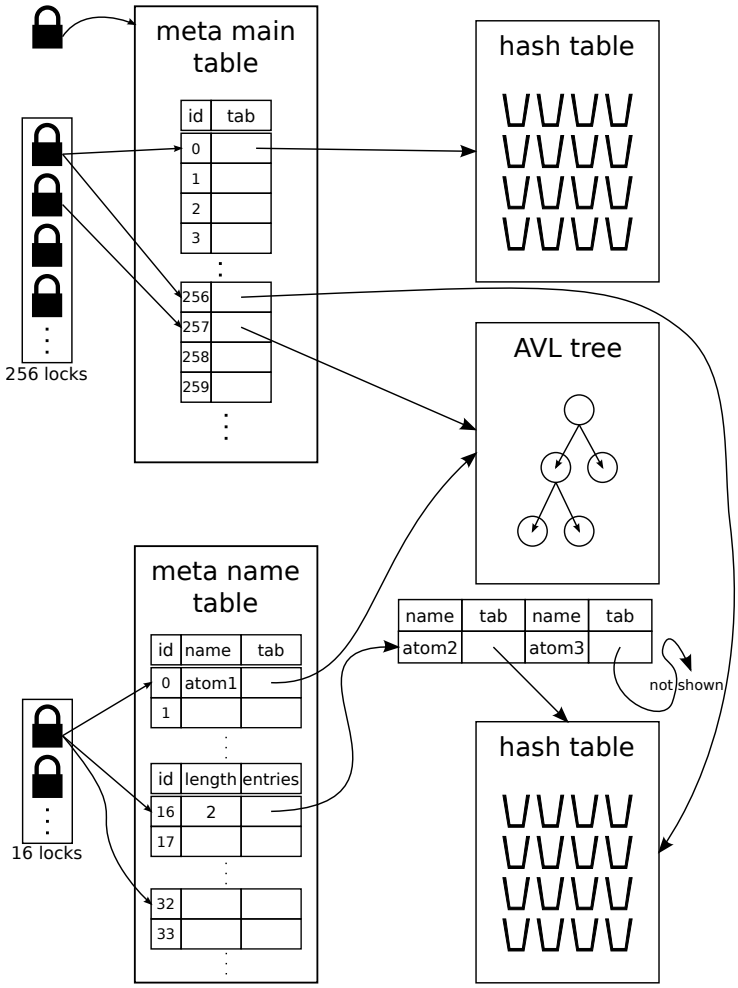


Figure 1: ETS meta tables: In this example, there are three ETS tables. An unnamed table with TID 0, and two named tables with TIDs 257 and 256. The tables with TIDs 0 and 256 use hashing, while the table with TID 257 is based on an AVL tree. TIDs 0 and 256 use the same lock in the `meta_main_table`. 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.

The following data structures are maintained on a node-wide level and are used for generic bookkeeping 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 runtime. 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.

3.2 Improvements Between Erlang/OTP Releases

ETS support for scalable parallelism has evolved over time, both before the start of the RELEASE project and, more importantly, during the project's duration. Here we briefly describe the major changes across Erlang/OTP releases.

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 was not so good.

3.2.1 Support for Fine Grained Locking

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`.

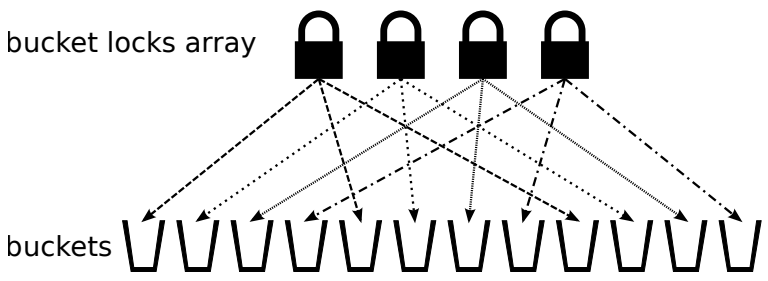


Figure 2: Mapping of locks to buckets using an array of four locks.

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 the option `write_concurrency` is enabled compared to just acquiring the table lock for reading when this option was not available. Thus enabling this 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.)

3.2.2 Reader Groups for Increased Read Concurrency

All shared 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 [11]. Transferring memory between private caches of cores is particularly expensive on Non-Uniform Memory Access (NUMA) systems, where cores can be located on 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 VM. A scheduler indicates a read by incrementing a counter in the memory area allocated for its reading group.

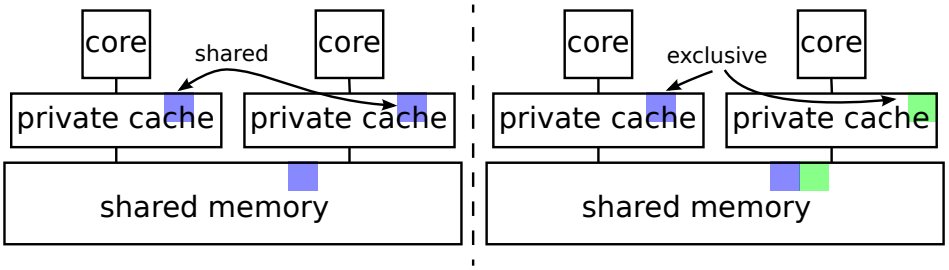


Figure 3: Illustration of cache line invalidation: When there is only a single counter (left, counter in blue), it can only be written from one core efficiently. It is invalidated in all other cores' caches on update. Using multiple counters in separate cache lines (right, counters in blue and green)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

3.2.3 More Refined Locking and More Reader Groups

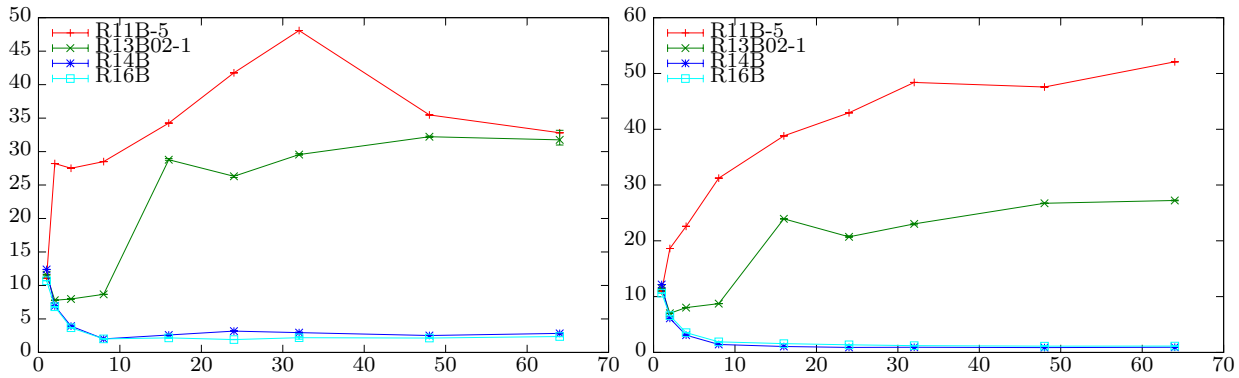
As mentioned in Section 3.2.1, the `write_concurrency` option enables fine grained locking for the hash based tables. Before R16B the buckets in the hash table were divided between 16 buckets. The size of the lock array was increased in Erlang/OTP R16B from 16 to 64 to give better scalability on machines 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 3.3.2 and 3.4.1.

3.3 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, 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. For the compilation of the Erlang/OTP system, 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 [3]. 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 in the range [1..2 097 152]. To make use of the available processing resources, the benchmark starts one worker process per



(a) Workload with 90% lookups and 10% updates. (b) Workload with 99% lookups and 1% updates.

Figure 4: Scalability of ETS tables of type `set` across Erlang/OTP releases.

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% and 99% 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 of this section, the x-axis shows the number of runtime schedulers and the y-axis shows runtime 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 runtime of these three runs. We also show the minimum and maximum runtime for the three runs as a vertical line at each data point.

3.3.1 Scalability of Hash-Based ETS Tables Across OTP Releases

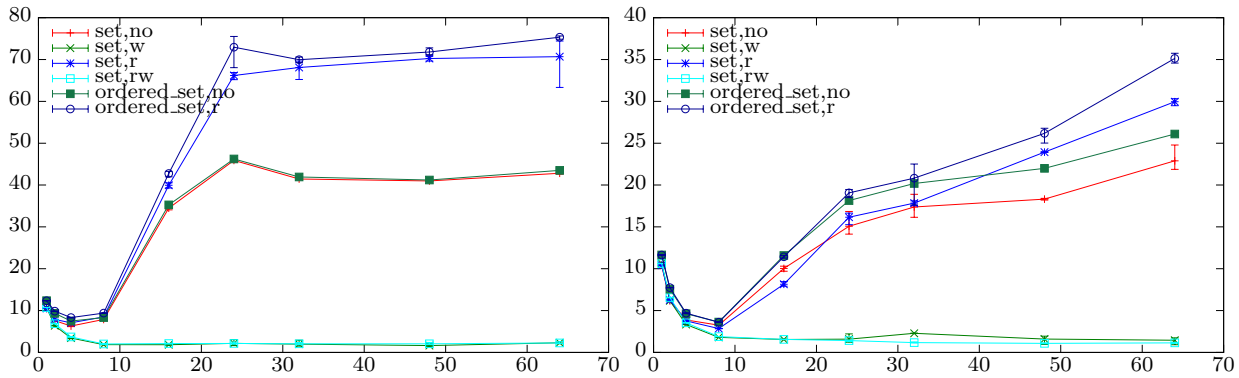
First, we measure scalability of the Erlang/OTP releases with major changes in the implementation of ETS (R11B-5, R13B02-1, R14B, and R16B), for accessing tables of type `set` using the workloads mentioned above. The results are shown in Figure 4.

As can be seen in Figure 4a, the scalability of ETS tables of type `set` improved significantly starting with release R14B. As described in Section 3.2, 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 `meta_main_table`), it is reasonable to believe, after looking at Figure 4a, that doing this is enough to give unsatisfactory scalability.

The results for the workload workload with 99% lookups and 1% updates, shown in Figure 4b, are very similar to those with 10% update operations. Compare them with those in Figure 4a.

3.3.2 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 3.2. Here we report on their effectiveness by measuring



(a) Workload with 90% lookups and 10% updates. (b) Workload with 99% lookups and 1% updates.

Figure 5: Scalability of ETS tables of type `set` when varying the concurrency options.

the performance effect for enabling vs. not enabling these options on the ets_bench benchmark run on Erlang/OTP R16B.

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

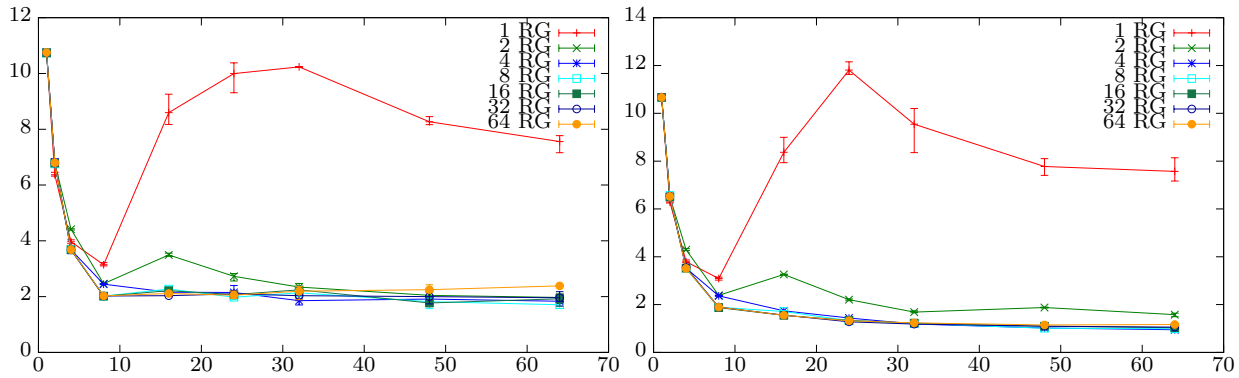
	no	r	w	r & w
set lock type	*normal*	*freq_r*	*freq_r*	*freq_r*
set bucket lock type	-	-	*normal*	*freq_r*
ordered_set lock type	*normal*	*freq_r*	*normal*	*freq_r*

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 reader groups and *freq_r* means the reader-writer lock with the reader groups optimization.

Figure 5 shows the performance results. Recall that the x-axis shows the number of schedulers and the y-axis shows runtime 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%.

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 options are enabled.

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 Figure 6. It is very clear 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 besed on hashing. It is worth noting that for this kind of ETS tables, 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.



(a) Workload with 90% lookups and 10% updates. (b) Workload with 99% lookups and 1% updates.

Figure 6: Scalability of ETS tables of type `set` when varying the number of reader groups.

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 hash-based 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 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.

3.4 Towards ETS with Even 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's VM. These ideas range from allowing more programmer control over the number of bucket locks in hash-based tables (Section 3.4.1), using contention-adapting trees to get better scalability for ETS tables of type `ordered_set` (Section 3.4.2), using more scalable locking libraries (Section 3.4.3), to adopting schemes for completely eliminating the locks in the meta table (Section 3.4.4).

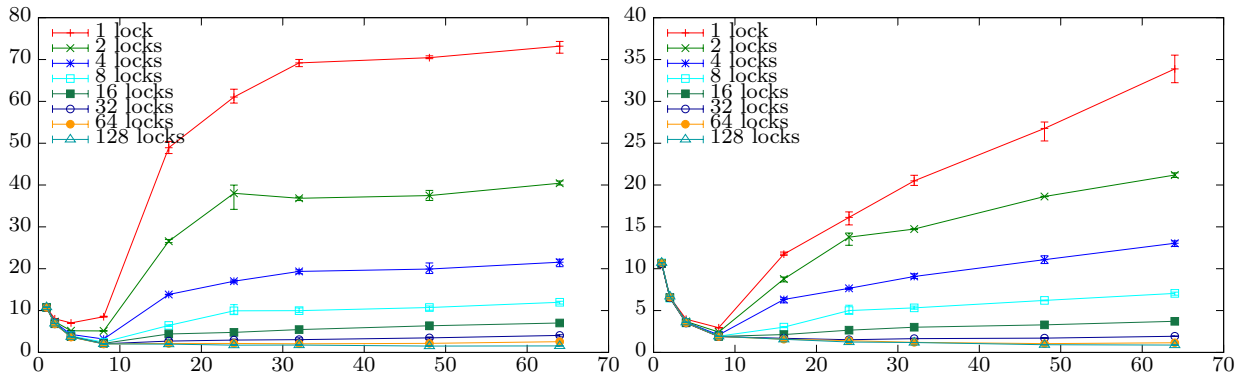
3.4.1 Number of Bucket Locks

As mentioned in Section 3.2.3 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.

Figure 7 shows the benchmark results for the two 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.

Since what is a good trade off depends on the application, our recommendation is to add support for setting the number of bucket locks at runtime and per table in future Erlang/OTP releases.



(a) Workload with 90% lookups and 10% updates. (b) Workload with 99% lookups and 1% updates.

Figure 7: Effect of the number of bucket locks on a table of type `set`.

3.4.2 Contention Adapting Trees for Ordered Set ETS Tables

As mentioned, the current implementation of ETS tables of type `ordered_set` is based on the AVL tree data structure. Every such shared table is protected by a single readers-writer lock which, unsurprisingly, is a scalability bottleneck in situations where there are multiple writers to the table, even when the percentage of write operations is very low (only 1% of the total table operations) [13].

To improve the scalability of ETS tables of type `ordered_set` we have proposed the use of *contention adapting search trees* or *CA trees* for short. A CA tree [12] implements the abstract data type ordered set and supports operations such as insert, delete and lookup as well as efficient ordered traversal. 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.

A brief description of CA trees 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 8 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 tree's root.

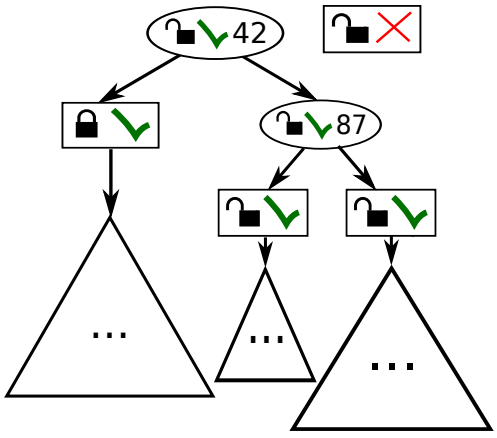


Figure 8: The structure of a CA tree.

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.

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 tree's root). 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.

Integrating CA trees into the Erlang Runtime System Two variants of the CA tree have been implemented and integrated into the Erlang VM as two new table types for testing purposes. One of these variants uses the *Treap* data structure [2] 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 the Erlang/OTP ETS implementation for `ordered_set`.

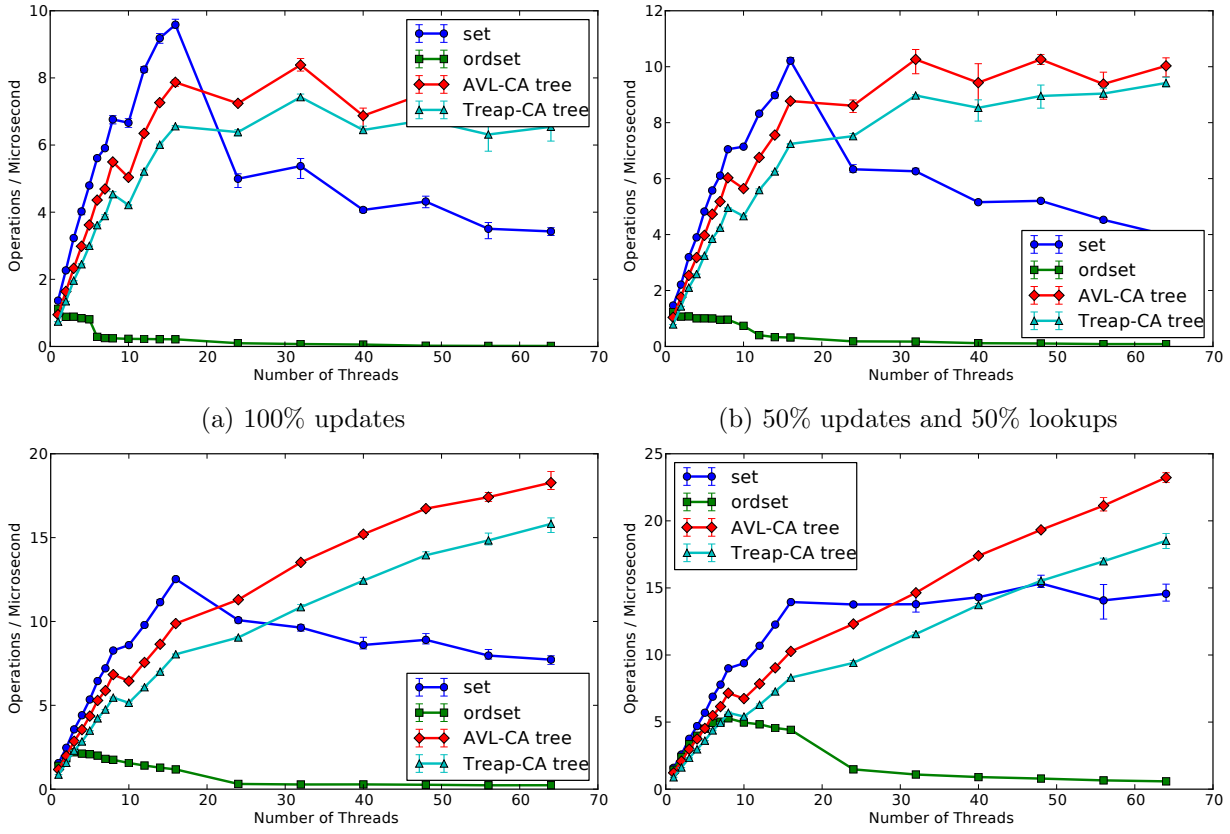
The CA tree integration is currently a prototype in the sense that it does not yet support the full interface of ETS. The operations currently supported are insert, delete, and lookup. However, extending the implementations to support the full ETS interface is quite easy [13]. For example, the operations 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. To see how operations from the ETS API that atomically operate on several keys can be implemented in CA trees and for memory management issues refer to a paper published at the Erlang Workshop [13].

A taste of CA trees' performance In the same paper [13], we conducted extensive performance and scalability evaluations for the new table types (AVL-based and Treap-based CA trees). We compared the new table types with the current implementations of `ordered_set` and `set` in various scenarios. We refer the interested reader to that paper for the complete set of the results, but we also include selected results here. The machine used is the same as before (see page 15), but we used Erlang/OTP 17.0 as a basis for our prototype and for comparison.

Figure 9 shows the scalability of the different ETS table variants as the number of schedulers (threads) increases. Note that in contrast to the previous graphs, the y-axis of these graphs shows throughput (i.e., the higher the better), not runtime. The results show the CA trees provide very good scalability (esp. compared with the current implementation of `ordered_set`) and also 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.x) 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.

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. The code needs to be refactored to conform to Erlang/OTP runtime system's coding conventions. More importantly, support for the full ETS API needs to be implemented, but, as explained, we expect that this will be a relatively easy task.

If it is decided to integrate CA trees into ETS, one also has to choose the way that this 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



(a) 100% updates

(b) 50% updates and 50% lookups

(c) 10% updates and 90% lookups

(d) 1% updates and 99% lookups

Figure 9: Scalability of the CA tree variants compared to `ordered_set` and `set` in Erlang/OTP 17.0.

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.

3.4.3 More Scalable Locking Libraries for ETS

We have also experimented and proposed the use of a *queue delegation locking* library [7] for the locks used in the implementation of ETS. *Queue Delegation (QD) locking* [8], is a new efficient delegation algorithm whose idea is simple. When, e.g., a shared data structure protected by a single lock is contended, the threads do not wait for the lock to be released. Instead, they try to delegate their operation to the thread currently holding the lock (called the *helper*). If the operation does not read from the shared structure, successful delegation allows a thread to immediately continue execution, possibly delegating more operations or doing other work. The helper thread is responsible for eventually executing delegated operations in the order they arrived to ensure linearizability. Most kinds of critical sections can be delegated with this scheme, and waiting is only needed when effects of a critical section need to be visible. We refer the reader to the paper [8] describing QD locking in detail for more information on its implementation and its properties. Below we describe the steps we went through when porting the ETS code to use an MRQD (Multiple Readers Queue Delegation) lock instead of a readers-writer lock and the performance we got.

Porting The porting focussed on the ETS operations `insert`, `delete` and `lookup`. The first two operations are interesting since they do not have any return value and can thus be delegated to the current lock holder without any need for the process issuing them to wait for their execution. On

the other hand, the `lookup` operation needs to wait for its result. We divided the porting work into three steps of increasing difficulty, where each step produced working code that we could benchmark to measure the resulting performance. We started from an ETS code base of eight files with a total of 16 277 lines of code.

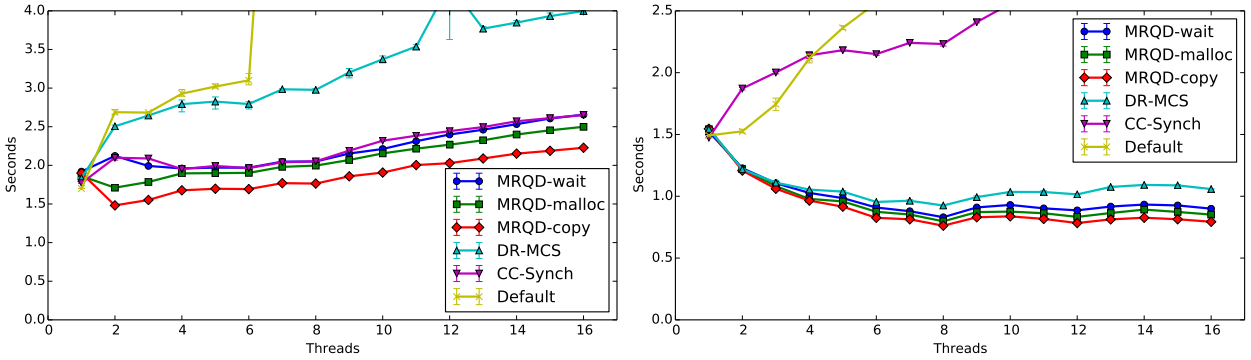
1. (delegate and wait) In this step we just delegate the original critical section and wait for its actual execution with the `LL_delegate_wait` function from the C queue delegation library. Usually this works without any semantic change of the original code. However, if thread-local variables are accessed inside the critical section, as was the case in the ETS code, care must be taken so that the right thread-local variable is accessed. In the ETS code, the thread-local variable access was subtle since it was done in the read-unlock call of a readers-writer lock. To fix this issue we simply moved the read-unlock call after the issuing of the critical section. Another way to deal with this problem would have been to pass a reference to the thread-local variable to the delegated function. In total this step required changing about 400 lines of code (60 of which were changes and 340 were additions, many to integrate with the existing locking structure).

2. (delegate without wait) To delegate without waiting for the actual execution of the critical section required more changes. The original code did some checking of parameters inside the critical section that could result in a return value indicating an error. These checks did not need to be done inside the critical section and could simply be lifted out. The parameters to the `insert` and `delete` functions are allocated on the heap of the issuing process and can be deallocated as soon as the functions have returned. Therefore it is not safe to send references to these values to delegated critical sections. Instead, we changed the original code to allocate a clone of the value and send a reference to the clone. For the `insert` case, the clone is in a form that can be inserted directly into the table data structure. The effort of allocating a clone is therefore not wasted since a clone would need to be created anyway to store the object. Furthermore, since the cloning is done outside the critical section, this modification can also decrease the length of the critical section. However, if the object being inserted is replacing an existing larger object, the original code had less memory management cost because it would just overwrite the existing object. For the `delete` operation, both the allocation of the cloned key and its subsequent freeing incurs an overhead compared to the original code. This step required changes in about 400 more lines of code (760 if one starts counting the differences from the original code).

3. (delegate and copy directly into the QD queue) In this step we got rid of the need to do more memory management than the original code by copying all parameters needed in the critical section directly into the queue buffer of the QD lock. This also had the benefit of improving the cache locality for the helper thread that is executing the critical sections. The only additional porting effort required in this step was the serialization of the key and the object to a form that can be stored directly into the QD queue. This step required changing only about 100 more lines of code.

Performance evaluation Once again, we used ets_bench to evaluate the performance and scalability of ETS tables of type `ordered_set` after applying each porting step described in the previous section. As mentioned, ets_bench measures the performance of ETS under variable contention levels and distributions of operations. We ran the benchmark on the machine we described on page 15 and all code was compiled using GCC version 4.7.2 with -O3. We pinned the software threads to logical processors so that the first eight software threads in the graphs were pinned to separate cores. Each configuration was run three times and we report the average run time.

The update only scenario presented in Figure 10a shows the run time of N Erlang processes performing $2^{22}/N$ operations each. The inserted objects are Erlang tuples with an integer key randomly selected from the range $[0, 2^{16}]$. The operations are `insert` or `delete` with equal probability.



(a) 100% updates

(b) 80% lookups and 20% updates

Figure 10: Scalability benchmarks for ETS. Dataset size is 2^{16}.

The line labeled Default represents the readers-writer lock currently used by ETS. It is optimized for frequent reads and the uncontended case. Therefore, it does not scale well with parallel writers. We also include the state-of-the-art readers-writer lock DR-MCS presented by Calciu *et al.* [4]. The MCS lock [10] that DR-MCS uses to synchronize writers is good at minimizing cache coherence traffic in the lock hand over. However since MCS is a queue-based lock, the thread that executes critical sections is likely to alternate between the cores. This is causing a lot of expensive cache coherence traffic inside the critical section which is one reason why its performance is worse than all delegation-based locks.

CC-Synch [6] is included in our comparison to show an alternative delegation locking mechanism that can be used with the same interface as QD locking. MRQD-copy, which is corresponding to porting step 3 in the previous section, performs best in all contended cases. MRQD-copy is closely followed by MRQD-wait, CC-Synch (both step 1) and MRQD-malloc (step 2). MRQD-wait, CC-Synch and MRQD-malloc are almost indistinguishable except for the case with two threads. MRQD-malloc performs better in this case because of its ability to continue directly after delegating work.

In Figure 10b we show the performance when 80% of the operations are `lookup`s and the rest are `insert`s and `delete`s with equal probability. Unsurprisingly, since they use the same read synchronization algorithm, the order between MRQD-copy, MRQD-malloc, MRQD-wait and DR-MCS is the same as in the update only scenario. With 99% `lookup` operations the difference between MRQD-copy, MRQD-malloc, MRQD-wait and DR-MCS is very small, but the performance advantage of the four delegation-based algorithms gets larger and larger with more updates.

3.4.4 More Scalable Alternatives to Meta Table Locking

As described in Section 3.1, 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 3.2.2 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 pointers 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.

4 Scalable Tracing Support for Profiling and Monitoring

This section presents the results of our work in providing improved support for DTrace/SystemTap in Erlang/OTP. It is divided into two parts: one describing probes related to the back-end for both offline and online profiling tools (such as Percept and Devo), and one describing probes specific for SD Erlang. The results of our early work on this front were presented in Deliverable D2.2 and most of the implemented probes were introduced in Deliverable D5.2, alongside with the description of the profiling tools — they are repeated here, as they can be useful as parts of the VM, not tied to any specific set of profiling tools.

4.1 DTrace/SystemTap back-end for offline/online profiling and monitoring

In the first year of the project, we designed, implemented and delivered the first version of a new back-end for *Percept* that uses DTrace or SystemTap to collect the information that *Percept*'s current back-end collects using Erlang built-in tracing and profiling mechanism. Since then, this back-end was enhanced, primarily in order to be able to profile distributed applications according to the revised architecture that is shown in Figure 7 of the revised Deliverable D5.1.

A number of profile and trace flags were introduced, so that users can specify the events that they need to trace. Each of these flags is mapped to a set of DTrace probes that need to be enabled. So, based on the specified profile and trace flags, the D or SystemTap script, which is to be executed on the application nodes, is generated dynamically. While profiling an application, the user can at any point start tracing one or more application nodes, stop tracing one or more other nodes, or modify the events that they need to trace. The back-end was extended to be used in *Percept*'s extension: *Percept2*.

A number of new probes were introduced after the first year of the project, to enable back-ends for online profiling and monitoring tools. As mentioned in the relevant section of Deliverable D5.2, the events that were to be traced by such tools are: process migrations, run queue sizes, inter-node message passing and events related to SD Erlang. Apart from the latter, which are described in detail in Section 4.2, the probes required to support these events are described below. For each probe, the description contains the event that causes it, the names and the types of its parameters, and the kind of information that is passed to them.

Probe: run_queue-create

Fired: whenever a run queue is created

Header:

```
probe run_queue__create(int rqid)
```

Parameters:

- `rqid`: the ID of the run queue

Probe: run_queue-enqueue

Fired: whenever a process is added to a run queue

Header:

```
probe run_queue__enqueue(char *p, int priority, int rqid, int rqsize)
```

Parameters:

- `p`: the PID of the process
- `priority`: the priority of the process
- `rqid`: the ID of the run queue
- `rqsize`: the size of the run queue (after the addition)

Probe: run_queue-dequeue

Fired: whenever a process is removed from a run queue

Header:

```
probe run_queue__dequeue(char *p, int priority, int rqid, int rqsize)
```

Parameters:

- `p`: the PID of the process
- `priority`: the priority of the process
- `rqid`: the ID of the run queue
- `rqsize`: the size of the run queue (after the removal)

Probe: process-migrate

Fired: whenever a process migrates from one run queue to another

Header:

```
probe process__migrate(char *p, int priority, int fromrqid, int fromrqsize,
                       int torqid, int torqsize)
```

Parameters:

- `p`: the PID of the process
- `priority`: the priority of the process
- `fromrqid`: the ID of the run queue, from which the process was removed
- `fromrqsize`: the size of the run queue, from which the process was removed (after the removal)
- `torqid`: the ID of the run queue, to which the process was added
- `torqsize`: the size of the run queue, to which the process was added (after the addition)

By adding new DTrace probes in order to export from the runtime system information that was not supported by the existing probes (e.g., the run queue sizes), we avoided the use of any sampling techniques.

4.2 DTrace probes for SD Erlang

SD Erlang has been essentially designed as a tweaked version of Erlang's *kernel* application. This leads to the observation that tracing specific events that are related to s_groups implies tracing calls of specific Erlang functions. So, in order to trace s_group-related events using DTrace or SystemTap, we had two options.

Our first option was to use the `global__function__entry` probe, and to use our D or SystemTap script to filter out any irrelevant function calls.

Probe: global-function-entry

Fired: whenever an external function is called

Header:

```
probe global__function__entry(char *p, char *mfa, int depth, uint64_t ts)
```

Parameters:

- `p`: the PID of the caller
- `mfa`: the MFA for the called function
- `depth`: the stack depth
- `ts`: the timestamp (in microseconds)

Our second option was to use one or more of the 951 `user_trace__n*` "user" probes. The integer that follows `n` in the name of the probe is the *probe number*. These probes can be triggered from inside the `s_group` module, whenever something interesting happens (e.g., a new s_group is created). The user tag for all these probes is set to "s_group".

Probe: user_trace-n0

Fired: whenever any of the `pn` functions of the `dyntrace` module is invoked with 0 as its first argument

Header:

```
probe user_trace__n0(char *proc, char *user_tag, int i1, int i2, int i3, int i4,
                     char *s1, char *s2, char *s3, char *s4)
```

Parameters:

- `proc`: the PID of the process that fired the probe
- `user_tag`: a user tag
- `i1`: an integer
- `i2`: an integer
- `i3`: an integer
- `i4`: an integer
- `s1`: a string
- `s2`: a string
- `s3`: a string
- `s4`: a string

We finally chose to take advantage of the "user" probes, and moreover to connect each type of s_group-related events that we are interested in to one such probe. The correspondence between s_group events and probe numbers, as well as the information that these probes are expected to carry are shown in Table 2.

Event	Probe number	Arguments
Creation s_group	0	s1: the group name
Deletion of s_group	1	s1: the group name
Addition of a node into an s_group	2	s1: the group name, s2: the node name
Removal of a node from an s_group	3	s1: the group name, s2: the node name

Table 2: Correspondence between s_group events and probe numbers.

Note that based on the above table, the call of an s_group function might cause more than one probe to be fired. For example, a call to the `s_group:new_s_group/2` will fire one `user_trace-n0` probe for the creation of the new s_group, and one more `user_trace-n2` probe for each one of the nodes that are contained in the list that is passed as a second argument to this function call.

The advantages of using "user" probes to trace s_group events are that it does not involve any irrelevant probe firings, which need to be ignored, and that it allows us to have an even more fine-grained control on the s_group events we want to trace (e.g., we can trace only creations of s_groups). On the other hand, our approach has the disadvantage that, since at the moment there is no way to reserve a specific probe label, anyone can end up using the same probe number to trace some other type of events.

5 Efficient Support for Benchmarking and Profiling Sim-Diasca

Sim-Diasca is a discrete-time simulation engine developed by EDF and designed to be applied to large-scale complex systems. One of the goals of RELEASE is to demonstrate how Erlang applications scale on clusters of multicore machines, how the scalability improvements of the Erlang VM and the command-line options that it offers affect the performance of these applications, and how tools developed by the RELEASE project can be used to identify bottlenecks in these applications. This section describes the actions that needed to be taken in order to explore the internals of Sim-Diasca using two of the tools that have been developed by RELEASE: BenchErl and Percept2.

5.1 BenchErl

BenchErl [3] is a scalability benchmark suite for applications written in Erlang, developed by RELEASE and described in Deliverable D2.1. As a first step in detecting potential problems in the implementation of Sim-Diasca, BenchErl was used in order to collect information about how Sim-Diasca scales with more VM schedulers, more hosts (i.e. computing nodes), different OTP releases, or different command-line arguments. But, in order to run Sim-Diasca from BenchErl, we first needed to overcome the following two problems:

- that the launching of all user and computing nodes was part of the application code, and
- that most of the parameters we wanted BenchErl to be able to control and vary were either hard-coded in some piece of Erlang code or hidden in some BASH script.

Essentially, what we did was move all the code that had to do with node launching from Sim-Diasca to BenchErl, and then prevent Sim-Diasca form either launching any nodes itself or shutting down any existing nodes that it finds. With these changes, the latest version of Sim-Diasca (2.2.8) is in the list of available benchmarks that one can use from BenchErl.

5.2 Percept2

The second tool that was used with Sim-Diasca was Percept2, a ocncurrency profiling tool for Erlang applications. The main purpose for using Percept2 was to dig into the internals of Sim-Diasca by

collecting information during its execution: how many processes it spawns, how many ports it opens, how many messages are sent, etc. Unfortunately, although Sim-Diasca is a distributed application, the current version of Percept2 does not run in a distributed mode: Percept2 can only profile the node where it is started. So, in order to profile Sim-Diasca we needed to start Percept2 on each computing node. But we had to make sure that Percept2 was started neither too early (so it would not capture information that was not related to the actual simulation, but rather, for example, with its setup) nor too late (so it would not lose any information from the simulation execution). In order to achieve this, we made use of the plugin mechanism provided by Sim-Diasca: we wrote a plugin that, as soon as the simulation starts, goes to each of the computing nodes and starts Percept2, and, as soon as the simulation ends, it stops Percept2 on all computing nodes that are involved in the simulation execution. That way, we end up with one file for each computing node that we can later analyze and visualize using Percept2.

6 Scalability of the VM Across Erlang/OTP Releases

We also used BenchErl to measure how VM improvements in different Erlang/OTP releases during the duration of the RELEASE project have affected the scalability and performance characteristics of the system. Note that this refers to the VM and is the context of a single Erlang node. For our evaluation, we chose each of the major release of the Erlang/OTP system (R15B, R16B and 17.0) and, for each major, one of their minor releases (R15B02, R16B03-1 and 17.4). The machine used is the one described in Section 3.3. Recall that the machine has four chips (NUMA nodes) of eight physical (sixteen logical) cores each, allowing the VM of a single Erlang node to use up to 64 schedulers.

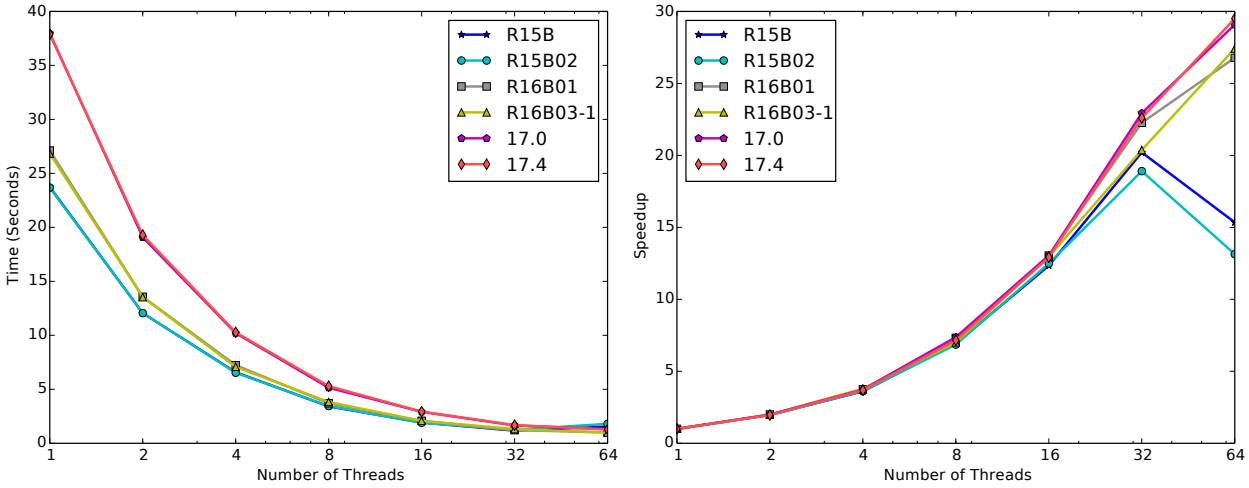
Results from a selected set of BenchErl benchmarks can be seen in Figures 11–15. Observe that the x-axis of all graphs uses a logarithmic scale. From these graphs, some general observations can be made:

- In absolute terms, the performance of the VM has occasionally (Figure 14a) though not always improved with time. To see the latter, cf. the lines in Figure 11a showing that the 17.x releases have worse performance than prior releases. A similar situation is shown in Figure 15a.
- On the other hand, what has clearly improved over time compared to the R15 (mainly) and R16 releases (to some extent) is the performance and scalability of the 17.x releases when the number of schedulers is high (16 and above). See for example the speedups graphs in Figures 11b and 12b but also the runtime graphs in Figures 12a, 14a and 15.
- Newer Erlang/OTP releases have become more effective in "protecting" themselves from performance degradation in cases where the scalability of applications is hindered for some reason, e.g., the hardware characteristics. In programs where it is difficult to achieve speedups beyond one NUMA node, such as pcmark (Figure 13a), the performance degradation as the processes run on more and more NUMA nodes is not as big in the 17.x releases compared to prior Erlang/OTP releases.

7 Concluding Remarks

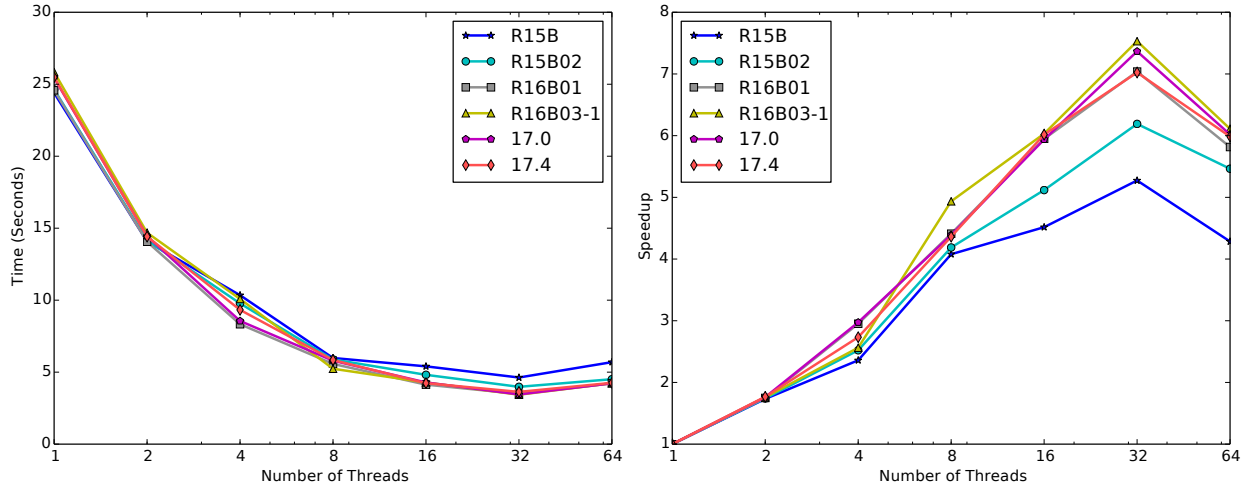
We have described changes and improvements to several key components of the Erlang Virtual Machine that have improved its robustness, scalability and responsiveness on big modern multicores. In short, we have presented:

- Two new additions to the Erlang Virtual Machine (VM), namely a scheduler utilization balancing mechanism and the ability to interrupt long running garbage collecting BIFs, that improve the responsiveness of the Erlang/OTP system.



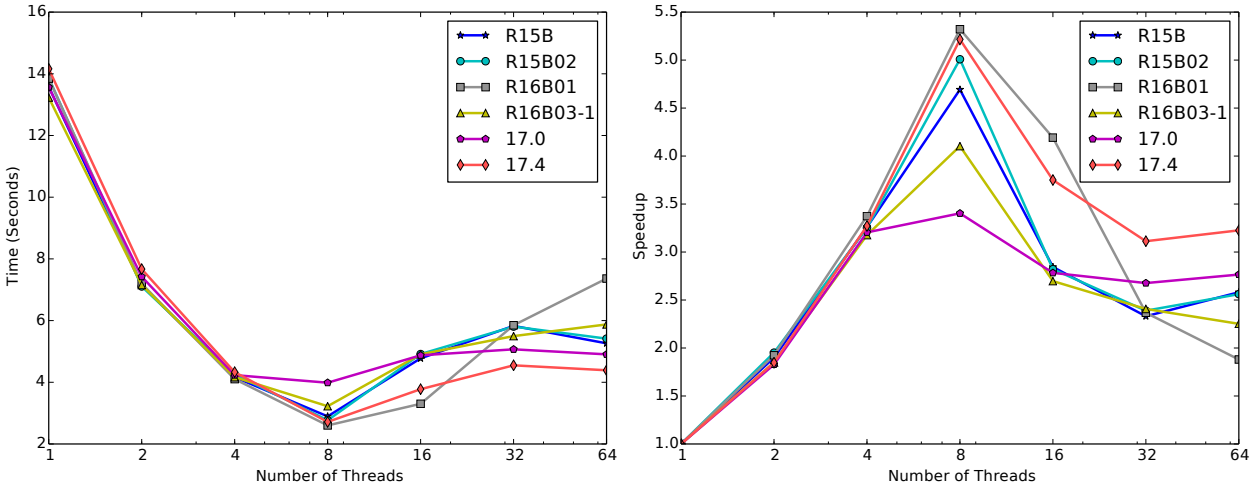
(a) mbrot runtime (b) mbrot speedup

Figure 11: Runtimes and speedups of different Erlang/OTP releases running the mbrot benchmark.



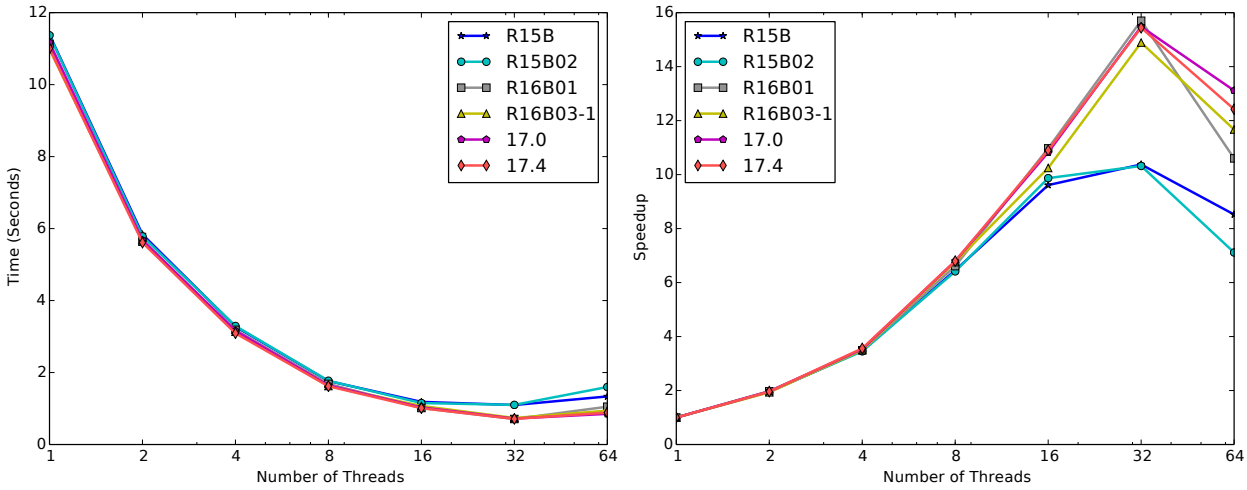
(a) moves runtime (b) moves speedup

Figure 12: Runtimes and speedups of different Erlang/OTP releases running the moves benchmark.



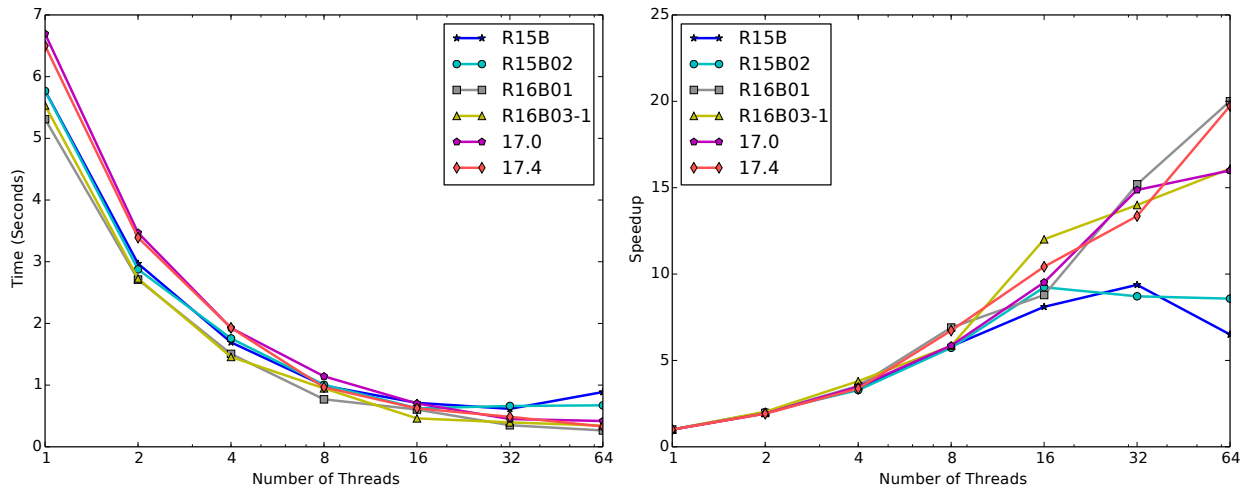
(a) pcmark runtime (b) pcmark speedup

Figure 13: Runtimes and speedups of different Erlang/OTP releases running the pcmark benchmark.

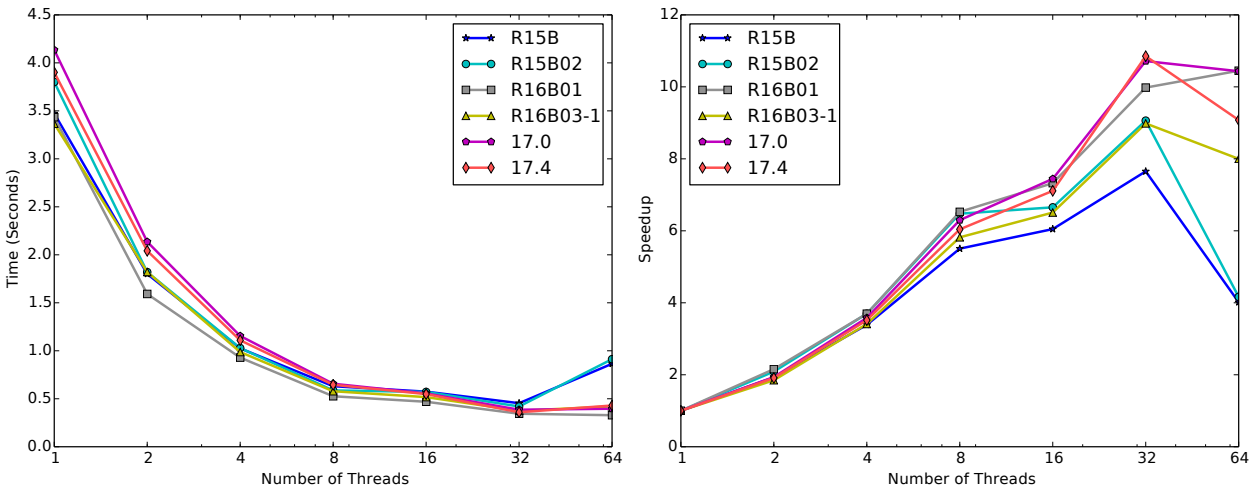


(a) ran runtime (b) ran speedup

Figure 14: Runtimes and speedups of different Erlang/OTP releases running the ran benchmark.



(a) orbit runtime without intra-worker parallelism (b) orbit speedup without intra-worker parallelism



(c) orbit runtime with intra-worker parallelism (d) orbit speedup with intra-worker parallelism

Figure 15: Runtimes and speedups of different Erlang/OTP releases running the orbit benchmark.

- Changes to the memory carrier migration mechanism, a new super carrier memory allocation scheme, and the new scheme for time management in the VM.
- A detailed description of the implementation of the Erlang Term Storage (ETS) and a study of the scalability and performance improvements to ETS across Erlang/OTP releases.
- New designs for ETS' implementation that improve its performance and scalability further.
- Scalable tracing support for profiling and monitoring SD Erlang applications.

Many of these changes have already found their place in an Erlang/OTP release (see also the information in the Appendix), and are used by the Erlang community. Most of the remaining changes currently exist in development branches of the system and are already scheduled to become part of an Erlang/OTP release in the near future. We hope that even those currently in prototype stage will soon also join them.

Change Log

Version	Date	Comments
0.1	17/3/2015	First Version Submitted to the Commission Services

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] 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. doi: 10.1109/SFCS.1989.63531. URL http://dx.doi.org/10.1109/SFCS.1989.63531.

[3] 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. doi: 10.1145/2364489.2364495. URL http://doi.acm.org/10.1145/2364489.2364495.

[4] 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. ACM, 2013. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442532. URL http://doi.acm.org/10.1145/2442516.2442532.

[5] CAtrees. CA Trees. http://www.it.uu.se/research/group/languages/software/ca_tree.

[6] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–266, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145849. URL http://doi.acm.org/10.1145/2145816.2145849.

[7] 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*, pages 572–583. Springer, 2014. Preprint available from http://www.it.uu.se/research/group/languages/software/qd_lock_lib.

[8] 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.

[9] 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.

[10] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991. ISSN 0734-2071. doi: 10.1145/103727.103729. URL http://doi.acm.org/10.1145/103727.103729.

[11] M. S. Papamarcos and J. 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. ISBN 0-8186-0538-3. doi: 10.1145/800015.808204. URL http://doi.acm.org/10.1145/800015.808204.

[12] K. Sagonas and K. Winblad. Contention adapting trees. Tech. Report, available in [5], 2014.

[13] K. Sagonas and K. Winblad. More scalable ordered set for ETS using adaptation. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, pages 3–11. ACM, Sept. 2014. ISBN 978-1-4503-3038-1. doi: 10.1145/2633448.2633455. URL http://doi.acm.org/10.1145/2633448.2633455.

A Virtual Machine Improvements in Erlang/OTP Releases

In this appendix we list changes and improvements that affect scalability, reliability and responsiveness of the Erlang Runtime System, which have made it into the Erlang/OTP system in one of its public releases during the duration of the RELEASE project. Naturally, major releases of the system (R15B, R16B, and 17.0) contain significantly many more changes and improvements than minor releases (R15B01, R15B02, R15B03, R16B01, R16B02, R16B03, 17.1, 17.3, and 17.4).

A.1 Improvements in Erlang/OTP R15B (2011-12-14)

- A number of memory allocation optimizations have been implemented. Most of them reduce contention caused by synchronization between threads during allocation and deallocation of memory. Most notably:
  - Synchronization of memory management in scheduler specific allocator instances has been rewritten to use lock-free data structures.
  - Synchronization of memory management in scheduler specific pre-allocators has been rewritten to use lock-free data structures.
  - The `mseg_alloc` memory segment allocator now uses scheduler specific instances instead of one global instance. Apart from reducing contention this also ensures that memory allocators always create memory segments on the local NUMA node on NUMA systems.
- The API of the `ethread` atomic memory operations used by the runtime system has been extended and improved. The library now also performs runtime tests for presence of hardware features, such as for example SSE2 instructions, instead of requiring this to be determined at compile time. All uses of the old deprecated atomic API in the runtime system have been replaced with the use of the new atomic API, a change which in many places implies a relaxation of memory barriers used.
- The Erlang Runtime System (ERTS) internal system block functionality has been replaced by new functionality for blocking the system. The old system block functionality had contention issues and complexity issues. The new functionality piggy-backs on thread progress tracking functionality needed by newly introduced lock-free synchronization in the runtime system. When the functionality for blocking the system is not used, there is practically no overhead. This since the functionality for tracking thread progress is there and needed anyway.
- An ERTS internal, generic, many to one, lock-free queue for communication between threads has been introduced. The many to one scenario is very common in ERTS, so it can be used in a lot of places in the future. Currently it is used by scheduling of certain jobs and the asynchronous thread pool, but more uses are planned for the future.
  - Drivers using the `driver_async` functionality are not automatically locked to the system anymore, and can be unloaded as any dynamically linked in driver.
  - Scheduling of ready asynchronous jobs is now also interleaved in between other jobs. Previously all ready asynchronous jobs were performed at once.
- The runtime system does not bind schedulers to logical processors by default anymore. The rationale for this change is the following: If the Erlang runtime system is the only operating system process that binds threads to logical processors, this improves the performance of the runtime system. However, if other operating system processes (as for example another Erlang runtime system) also bind threads to logical processors, there might be a performance degradation instead. In some cases this degradation might be severe. Due to this, there was a change in the default setting so that the user is required to make an active decision in order to bind schedulers.

A.2 Improvements in Erlang/OTP R15B01 (2012-04-02)

- Added `erlang:statistics(scheduler_wall_time)` to ensure correct determination of scheduler utilization. Measuring scheduler utilization is strongly preferred over CPU utilization, since CPU utilization gives very poor indications of actual scheduler/VM usage.

A.3 Improvements in Erlang/OTP R15B02 (2012-09-03)

- A new scheduler wake up strategy has been implemented. For more information see the documentation of the `+sws` command line argument of `erl`.
- A switch for configuration of busy wait length for scheduler threads has been added. For more information see the documentation of the `+sbwt` command line argument of `erl`.

A.4 Improvements in Erlang/OTP R15B03 (2012-12-06)

- The frequency with which sleeping schedulers are woken due to outstanding memory deallocation jobs has been reduced.

A.5 Improvements in Erlang/OTP R16B (2013-02-25)

- Various process optimizations have been implemented. The most notable of them are:
  - New internal process table implementation allowing for both parallel reads as well as writes. Especially read operations have become really cheap. This reduces contention in various situations (e.g., when spawning or terminating processes, sending messages, etc.)
  - Optimizations of run queue management reducing contention.
  - Optimizations of process state changes reducing contention.
- Non-blocking code loading. Earlier when an Erlang module was loaded, all other execution in the VM was halted while the load operation was carried out in single threaded mode. Now modules are loaded without blocking the VM. Processes may continue executing undisturbed in parallel during the entire load operation. The load operation is completed by making the loaded code visible to all processes in a consistent way with one single atomic instruction. Non-blocking code loading improves the real-time characteristics of applications when modules are loaded or upgraded on a running SMP system.
- Major port improvements. The most notable of them are:
  - New internal port table implementation allowing for both parallel reads as well as writes. Especially read operations have become really cheap. This reduce contention in various situations. For example when, creating ports, terminating ports, etc.
  - Dynamic allocation of port structures. This allows for a much larger maximum amount of ports allowed as a default. The previous default of 1024 has been raised to 65536. Maximum amount of ports can be set using the `+Q` command line flag of `erl`.
  - Major rewrite of scheduling of port tasks. Major benefits of the rewrite are reduced contention on run queue locks, and reduced amount of memory allocation operations needed. The rewrite was also necessary in order to make it possible to schedule signals from processes to ports.
  - Improved internal thread progress functionality for easy management of unmanaged threads. This improvement was necessary for the rewrite of the port task scheduling.
  - Rewrite of all process to port signal implementations in order to make it possible to schedule those operations. All port operations can now be scheduled which allows for reduced lock contention on the port lock as well as truly asynchronous communication with ports.

  - Optimized lookup of port handles from drivers.
  - Optimized driver lookup when creating ports.
  - Preemptable `erlang:ports/0` BIF.

  These changes imply changes of the characteristics of the system. The most notable are:

  Order of signal delivery. The previous implementations of the VM delivered signals from processes to ports in a synchronous fashion, which was stricter than required by the language. Starting with Erlang/OTP R16B, signals are truly asynchronously delivered. The order of signal delivery still adheres to the requirements of the language, but only to those. That is, some signal sequences that previously always were delivered in one specific order may now from time to time be delivered in different orders.

  Latency of signals sent from processes to ports. Signals from processes to ports where previously always delivered immediately. This kept latency for such communication to a minimum, but it could cause lock contention which was very expensive for the system as a whole. In order to keep this latency low also in the future, most signals from processes to ports are by default still delivered immediately as long as no conflicts occur. An example of such a conflict is not being able to acquire the port lock. When such conflicts occur, the signal will be scheduled for delivery at a later time. A scheduled signal delivery may cause a higher latency for this specific communication, but improves the overall performance of the system since it reduces lock contention between schedulers. The default behavior of only scheduling delivery of these signals on conflict can be changed by passing the `+spp` command line flag to `erl`. The behavior can also be changed on port basis using the parallelism option of the `open_port/2` BIF.

  Execution time of erlang:ports/0. Since the `erlang:ports/0` BIF now can be preempted, the responsiveness of the system as a whole has been improved. A call to `erlang:ports/0` may, however, take a much longer time to complete than before. How much longer time heavily depends on the system load.

  Reduction cost of calling driver callbacks. Calling a driver callback is quite costly. This was previously not reflected in reduction cost at all. Since the reduction cost now has increased, a process performing lots of direct driver calls will be scheduled out more frequently than before.

- The default reader group limit has been increased to 64 from 8. This limit can be set using the `+rg` command line argument of `erl`. This change of default value reduces lock contention on Erlang Term Storage (ETS) tables using the `read_concurrency` option at the expense of increased memory consumption when the amount of schedulers and logical processors are between 8 and 64.

- Increased potential concurrency in ETS for `write_concurrency` option. The number of internal table locks has been increased from 16 to 64. This makes it four times less likely that two concurrent processes writing to the same table would collide and thereby be serialized. The cost is an increased constant memory footprint for tables using `write_concurrency`. The memory consumption per inserted record is not affected. The increased footprint can be particularly large if `write_concurrency` is combined with `read_concurrency`.

- The scheduler wake up strategy implemented in Erlang/OTP R15B02 is now used by default. This strategy is not as quick to forget about previous overload as the previous strategy. This change implies changes of the system's characteristics. Most notable: When a small overload comes and then disappears repeatedly, the system will be willing to wake up schedulers for slightly longer time than before. Timing in the system will also change, due to this.

- The `+stbt` command line argument of `erl` was added. This argument can be used for trying to set scheduler bind type. Upon failure unbound schedulers will be used.

A.6 Improvements in Erlang/OTP R16B01 (2013-06-18)

- Introduced support for migration of memory carriers between memory allocator instances. This feature is not enabled by default and does not affect the characteristics of the system. However, when enabled, it has the effect of reducing memory footprint when the memory load is unevenly distributed between scheduler specific allocator instances.

A.7 Improvements in Erlang/OTP R16B02 (2013-09-18)

- New allocator strategy aoffcbf (*address order first fit, carrier best fit*). Supports carrier migration but with better CPU performance than aoffcaobf.

- Added command line option to set schedulers by percentages. For applications that show enhanced performance from the use of a non-default number of emulator scheduler threads, having to accurately set the right number of scheduler threads across multiple hosts each with different numbers of logical processors is difficult because the `+S` option requires absolute numbers of scheduler threads and scheduler threads online to be specified. To address this issue, a `+SP` command line option was added to `erl`, similar to the existing `+S` option but allowing the number of scheduler threads and scheduler threads online to be set as percentages of logical processors configured and logical processors available, respectively. For example, `+SP 50:25` sets the number of scheduler threads to 50% of the logical processors configured, and the number of scheduler threads online to 25% of the logical processors available. The `+SP` option also interacts with any settings specified with the `+S` option, such that the combination of options `+S 4:4 +SP 50:50` (in either order) results in two scheduler threads and two scheduler threads online.

A.8 Improvements in Erlang/OTP R16B03 (2013-12-09)

- A new memory allocation feature called *super carrier* has been introduced. The super carrier feature can be used in different ways. It can for example be used for pre-allocation of all memory that the runtime system should be able to use. By default the super carrier is disabled. It is enabled by passing the `+MMscs <size in MB>` command line argument.

A.9 Improvements in Erlang/OTP 17.0 (2014-04-07)

- Migration of memory carriers has been enabled by default on all Erlang runtime system internal memory allocators based on the `alloc_util` framework except for `temp_alloc`. That is, `+M<S>acul de` is default for these allocators. Note that this also implies changed allocation strategies for all allocators. They will all now use the *address order first fit carrier best fit* strategy introduced in Erlang/OTP R16B02. By passing `+Muacul 0` on the command line, all configuration changes made by this change can be reverted.

  This change improves the memory characteristics of the runtime system decreasing its memory footprint at the expense of a small performance cost.

- The garbage collection tenure rate has been increased. The garbage collector tries to maintain the previous heap block size during a minor garbage collection, i.e. 'need' is not utilized in determining the size of the new heap; instead the collector relies on tenure and garbage to be sufficiently large. Previously, in instances during intense growing with exclusively live data on the heap coupled with delayed tenure, full sweeps would be triggered directly after a minor garbage collection to make room for 'need' since the new heap would be full. To remedy this, the tenure of terms on the minor heap will always happen (if it is below the high watermark) instead of every other minor garbage collection.

  This change reduces the CPU time spent in garbage collection but may incur delays in collecting garbage from the heap.

- An experimental *dirty scheduler* functionality has been introduced. This functionality can be enabled by passing the command line argument `--enable-dirty-schedulers` to `configure` when building the system.

- Uses of `erlang:binary_to_term/1` now cost an appropriate amount of reductions and will interrupt the process executing the function call and yield to the scheduler if the term passed as their argument is big.

- Support for an LLVM backend has been added in the HiPE native code compiler.

A.10 Improvements in Erlang/OTP 17.1 (2014-06-24)

- The following native functions now cost an appropriate amount of reductions and yield the process to the scheduler when out of reductions:

  - `erlang:binary_to_list/1`
  - `erlang:binary_to_list/3`
  - `erlang:bitstring_to_list/1`
  - `erlang:list_to_binary/1`
  - `erlang:iolist_to_binary/1`
  - `erlang:list_to_bitstring/1`
  - `binary:list_to_bin/1`

  This change impacts:

  Performance: The functions converting from lists got a performance loss for very small lists, and a performance gain for very large lists.

  Priority: Previously a process executing one of these functions effectively got an unfair priority boost. This priority boost depended on the input size. The larger the input was, the larger the priority boost got. This unfair priority boost is now lost.

A.11 Improvements in Erlang/OTP 17.3 (2014-09-17)

- Introduced the `enif_schedule_nif()` function to the NIF API. This function allows a long-running NIF to be broken into separate NIF invocations without the help of a wrapper function written in Erlang. The NIF first executes part of the long-running task, then calls `enif_schedule_nif()` to schedule a NIF for later execution to continue the task. Any number of NIFs can be scheduled in this manner, one after another. Since the emulator regains control between invocations, this helps avoid problems caused by native code tying up scheduler threads for too long.

A.12 Improvements in Erlang/OTP 17.4 (2014-12-10)

- Introduced support for *eager check I/O*. When eager check I/O is enabled, schedulers will more frequently check for I/O work. Outstanding I/O operations will however not be prioritized to the same extent as when eager check I/O is disabled.

  By default eager check I/O is disabled and can be enabled using the `erl` command line argument `+secio true`. Eager check I/O impacts the following characteristics of the Erlang runtime system when enabled:

  - Results in lower latency and smoother management of externally triggered I/O operations.
  - Results in slightly reduced priority of externally triggered I/O operations.

- Optimization of atomic memory operations with release barrier semantics on 32-bit PowerPC.
- Improved support for atomic memory operations provided by the `libatomic_ops` library. Most importantly support for use of native double word atomics when implemented by `libatomic_ops` (for example, implemented for ARM).
- Minor adjustment of scheduler activation code making sure that an activation of a scheduler is not prevented by its run queue being non-empty.
- Improved allocation carrier migration search logic. This reduces the risk of failed migrations that could lead to excess memory consumption. It also improves SMP performance due to reduced memory contention on the migration pool.