Scalability and the Erlang VM

Kostis Sagonas

RELEASE Work Package 2 (VM) Leader

Overview

- Erlang and its concurrency model (very brief)
- Erlang VM and its implementation
- **BenchErl**: An extensible scalability benchmark suite for Erlang/OTP and its applications

  http://release.softlab.ntua.gr/bencherl
- A glimpse of the current scalability of Erlang/OTP

  + Demo
- A deeper look into ETS and its scalability

  + Demo

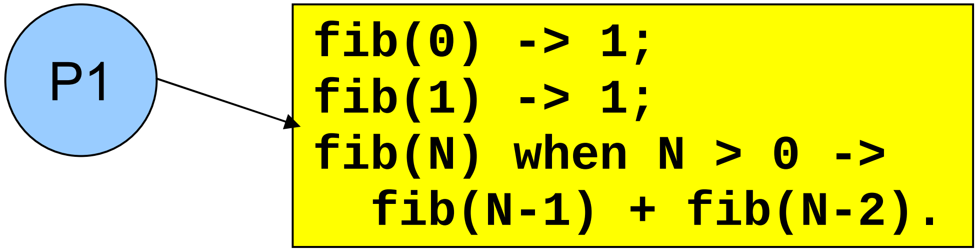
Erlang

- Functional
- Single-assignment
- Concurrent
- Distributed
- Message passing
- Soft real-time
- Fault tolerant
- No sharing (conceptually)
- Garbage collected
- Virtual Machine (BEAM)
- Native code (HiPE)
- Hot-swapping of code
- Multiprocessor support
- OTP libraries
- Open source

Why is Erlang Interesting?

- Appropriate for large scale concurrent execution
- Concurrency is built-in to the language
  - not an afterthought or provided via a library
- “Shared nothing” by default concurrency model
  - requires no explicit locking; in principle scales well
  - significantly eases concurrent programming!

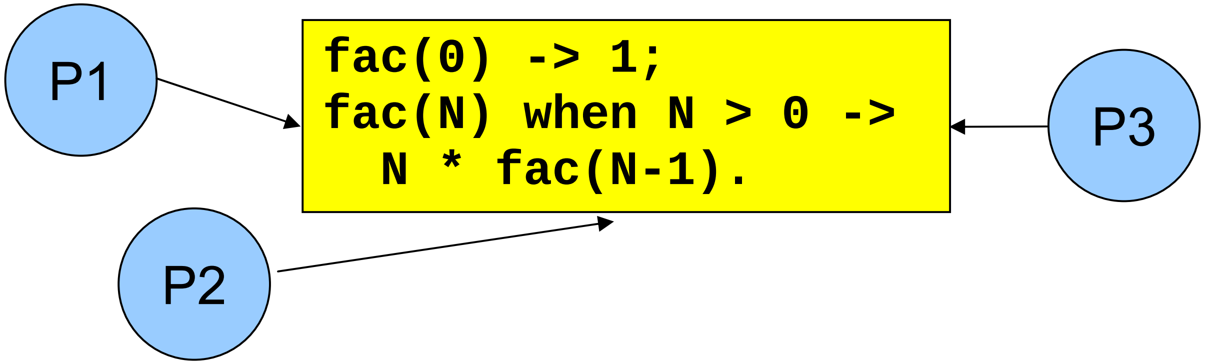
Erlang Concurrency



```
fib(0) -> 1;
fib(1) -> 1;
fib(N) when N > 0 ->
  fib(N-1) + fib(N-2).
```

- Whenever a program is running, the code is executed by a *process*
- A process keeps track of the current program point, the values of variables, the call stack, etc.
- Each process has a *unique process identifier* (“Pid”) that can be used to identify the process
- Processes are *concurrent* (they can run in parallel)

Concurrent Process Execution



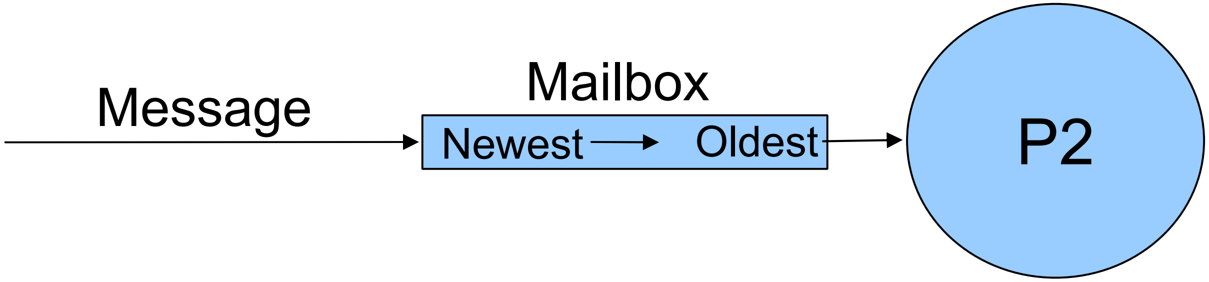
```
fac(0) -> 1;
fac(N) when N > 0 ->
  N * fac(N-1).
```

- Different processes may be reading the same program code at the same time
  - They have their own data, program point and stack – only the text of the program is being shared (almost...)
  - *The programmer does not have to think about other processes updating its state (variables)*
  - *No locking is needed* at the program level

Implementation of Concurrency

- Erlang processes are implemented by the VM's runtime system, not by OS threads
- Multitasking is *preemptive* (the VM does its own process switching and scheduling)
- Processes use very little memory initially, and switching between processes is very fast
- Erlang can handle very large numbers of processes
  - Some applications use more than 100K processes
- On multicores, processes can be scheduled in parallel on separate CPUs/cores

Message Passing and Queues



- Each process has a *message queue* (mailbox)
  - Arriving messages are placed in the queue
  - *No size limit* – messages are kept till extracted
- A process *receives* a message by *selectively* extracting it from the mailbox
  - Does not have to take the first message in the queue

Starting Processes

- The **spawn** functions create new processes

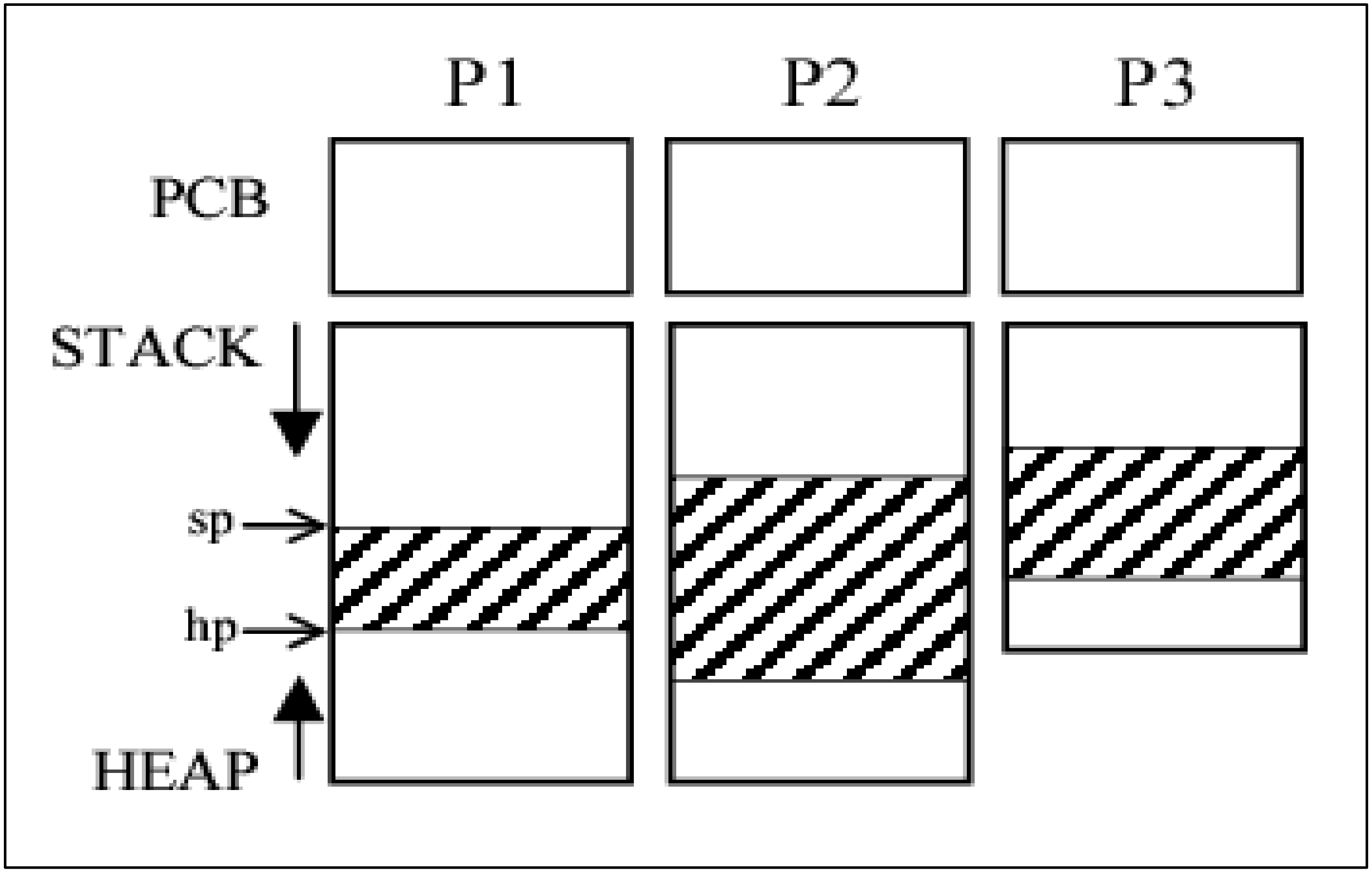
```
spawn(fun () -> ... end)
spawn(Module, Function, [A1,…,An])
```

- The new process will run the specified function
- The spawn operation always returns immediately
  - The return value is the Pid of the new process
  - The “parent” always knows the Pid of the “child”
  - The child will not know its parent until it's told

Erlang’s Runtime System (VM)

- Handles the basic “built-in” things:
  - Memory allocation
  - Garbage collection
  - Process creation
  - Message passing
  - Process scheduling
- Several possible ways of structuring the RTS
  - Trade-offs have been studied long ago
    - on single core architectures!

VM with Process Local Heaps



Process Local Heap Organization

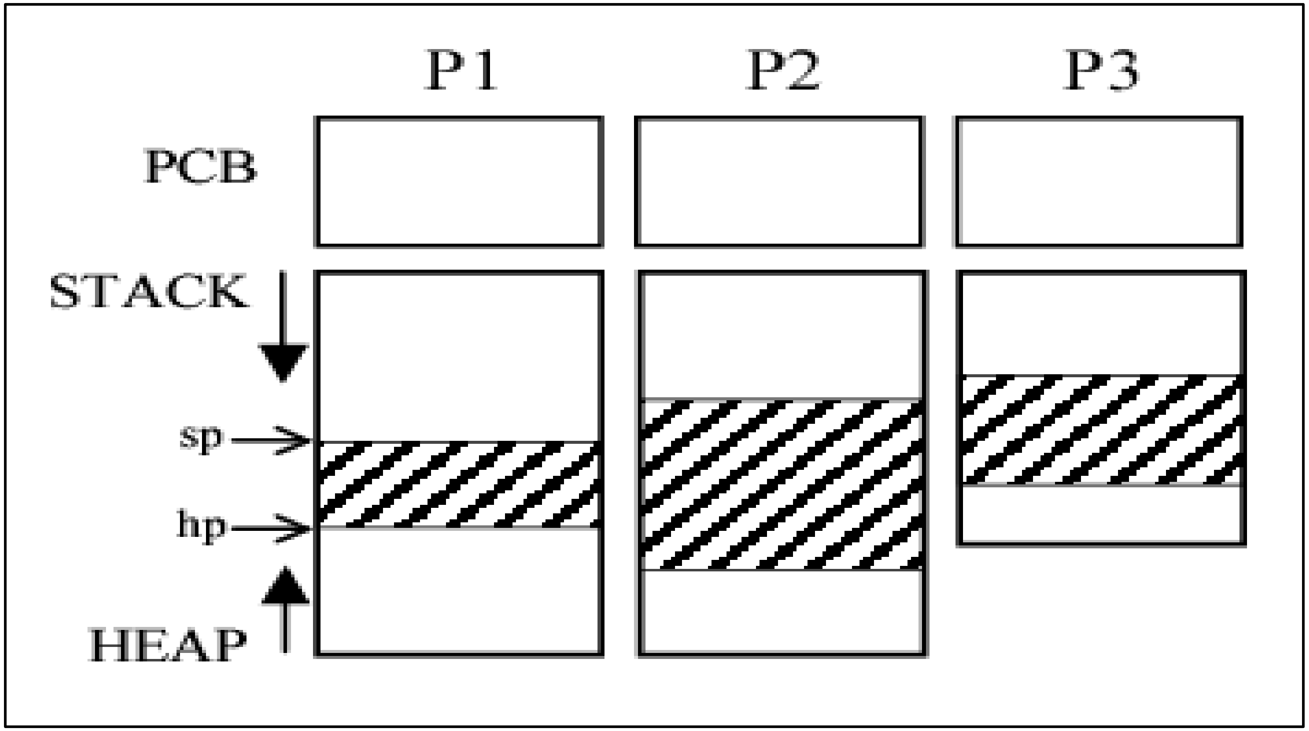
Pros:

- Isolation and robustness
- Processes can be GC-ed independently
- Fast memory deallocation when a process terminates: heaps are used as regions/arenas

Cons:

- Messages are always copied, even between processes on the same core (machine)
  - Sending is O(n) on the size of message
- Memory fragmentation is high(er)

The Truth About Erlang’s VM



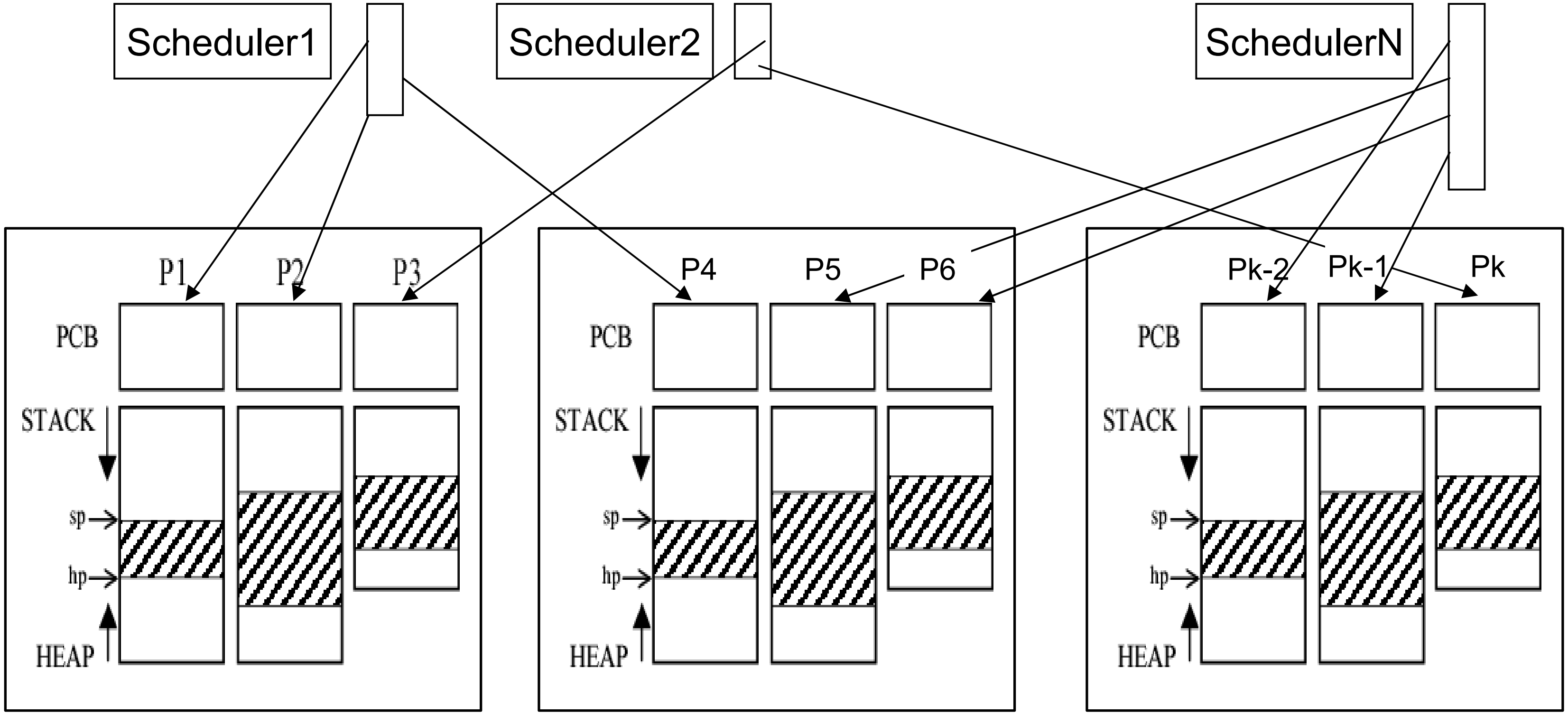
Global areas:

- Atom table
- Process registry

Erlang Term Storage

“Big Binary” Area

SMP Architecture



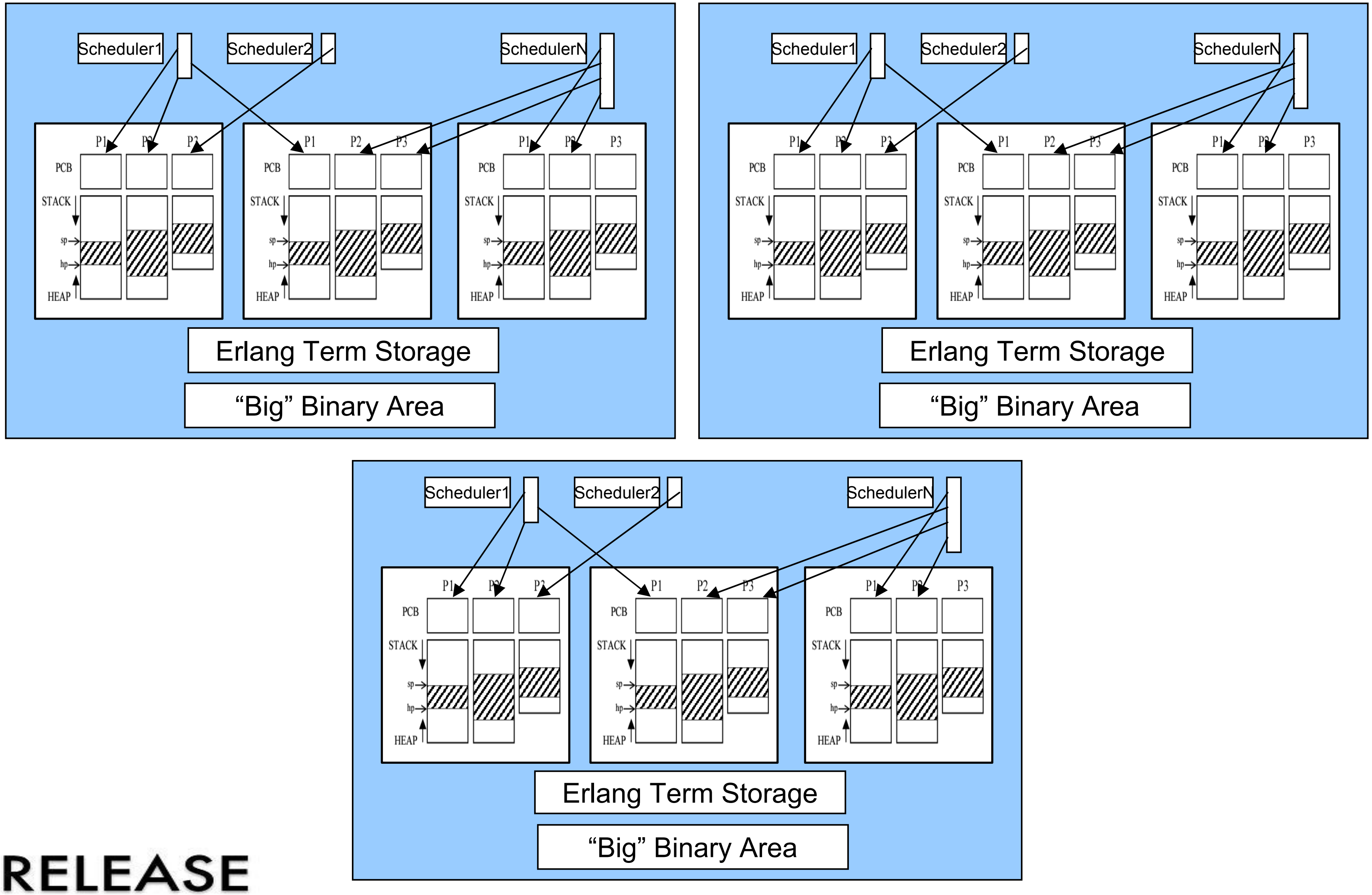
Global areas:

- Atom table
- Process registry

Erlang Term Storage

“Big Binary” Area

Distributed Architecture



Erlang Term Storage

“Big” Binary Area

Erlang Term Storage

“Big” Binary Area

Erlang Term Storage

“Big” Binary Area

Goals of RELEASE

Scale Erlang's concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines (100 000 cores).

- Language primitives for scalable distribution
- Virtual machine extensions and improvements
- Tools for parallelizing/refactoring existing code
- Tools for profiling and testing for errors
- Scalable virtualization infrastructure
- Porting Erlang/OTP on the Blue Gene

Download and Install BenchErl

http://release.softlab.ntua.gr/bencherl

```
git clone git@github.com:
          softlab-ntua/bencherl.git
cd bencherl
make
make ui
```

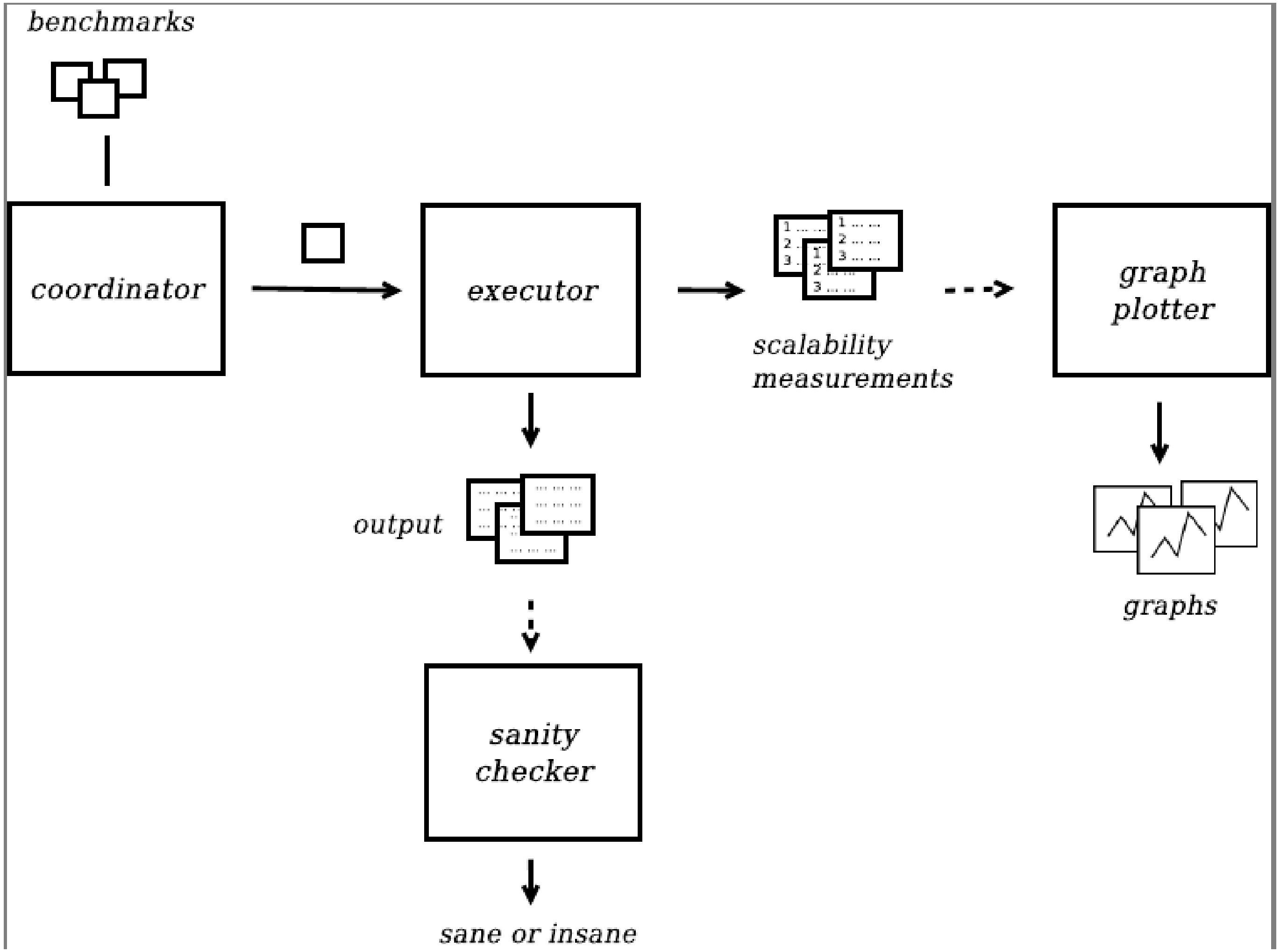
BenchErl: Benchmarking Suite

- Started with 14 concurrent programs used at Ericsson for benchmarking the performance of the SMP implementation of Erlang/OTP
- Added some bigger applications
- Infrastructure to measure performance of the system as the number of cores increases
- Shows scalability results on three platforms:
  - A Sandy-bridge i7 (4 cores, 8 hyperthreads)
  - A 4-year old Xeon server with 16 cores
  - An AMD “Bulldozer”-based server with 64 cores

BenchErl

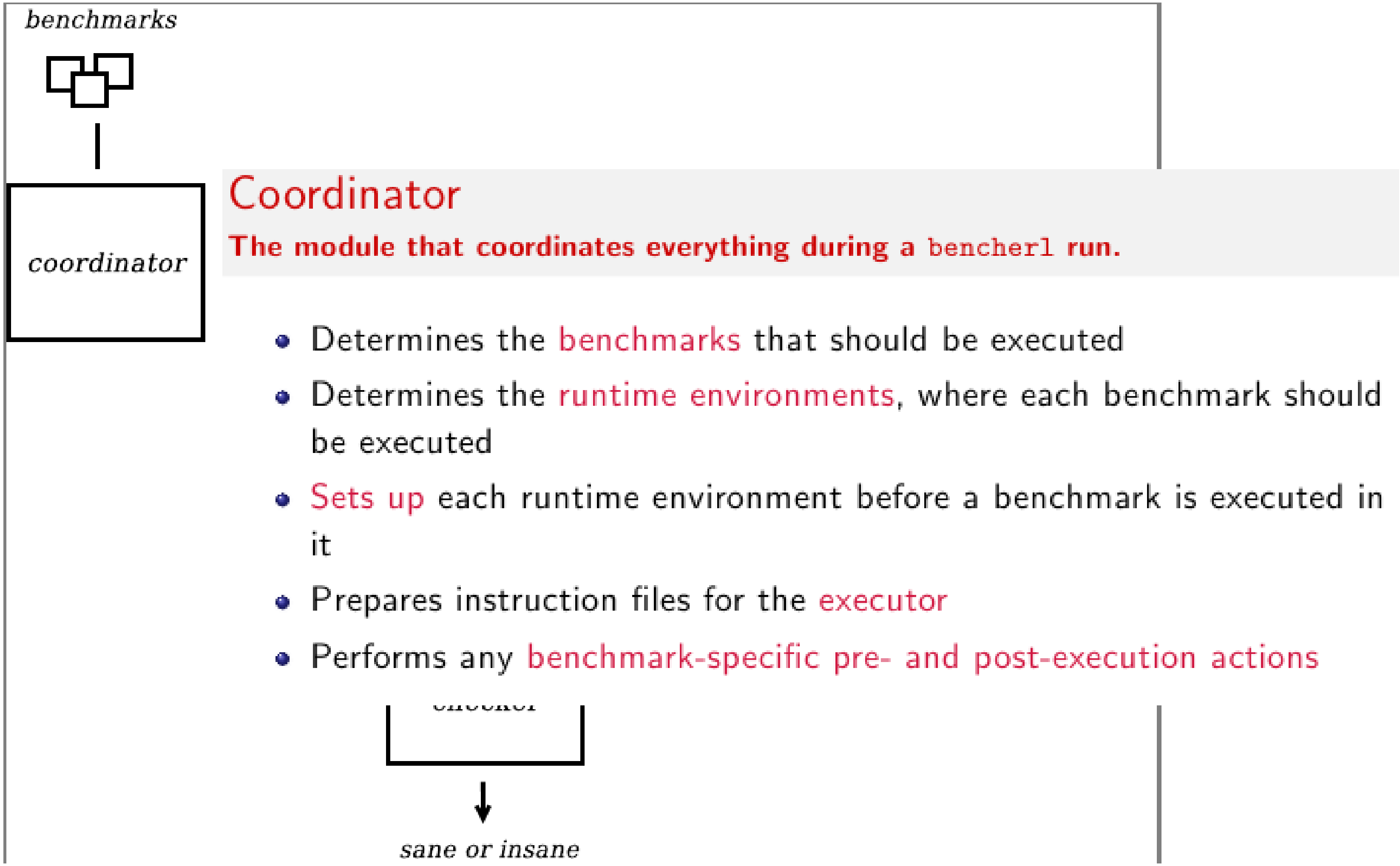
- Serves both as
  - a tool to run and analyze benchmarks
  - an extensible benchmark repository
- Focuses on *scalability* w.r.t. the following
  - *number of Erlang nodes*
  - *number of CPU cores*
  - *number of schedulers*
  - *Erlang/OTP release and flavor*
  - *settings of various command-line arguments*

BenchErl's Architecture



RELEASE

BenchErl’s Architecture

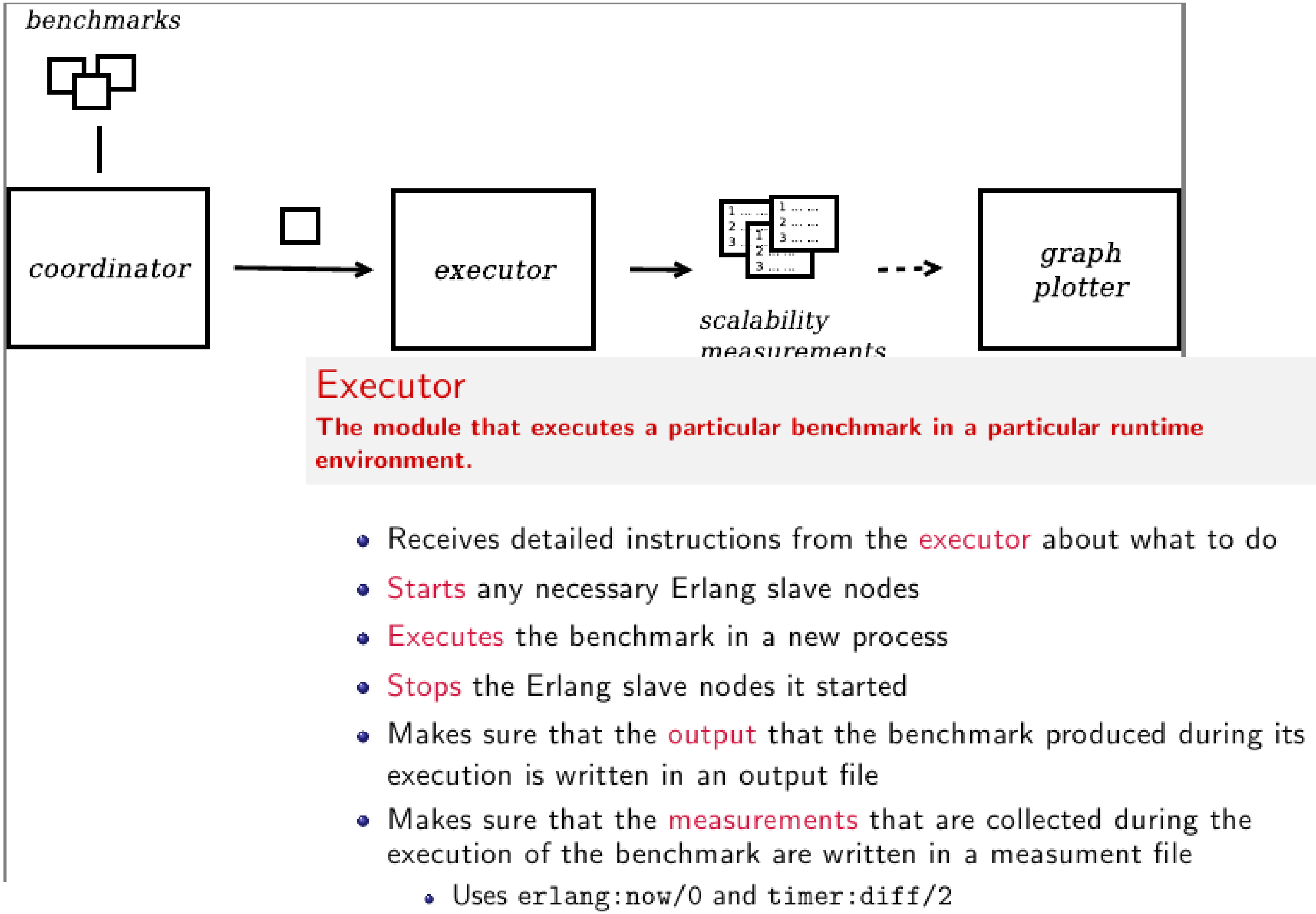


Coordinator

The module that coordinates everything during a `bencherl` run.

- Determines the benchmarks that should be executed
- Determines the runtime environments, where each benchmark should be executed
- Sets up each runtime environment before a benchmark is executed in it
- Prepares instruction files for the executor
- Performs any benchmark-specific pre- and post-execution actions

BenchErl’s Architecture



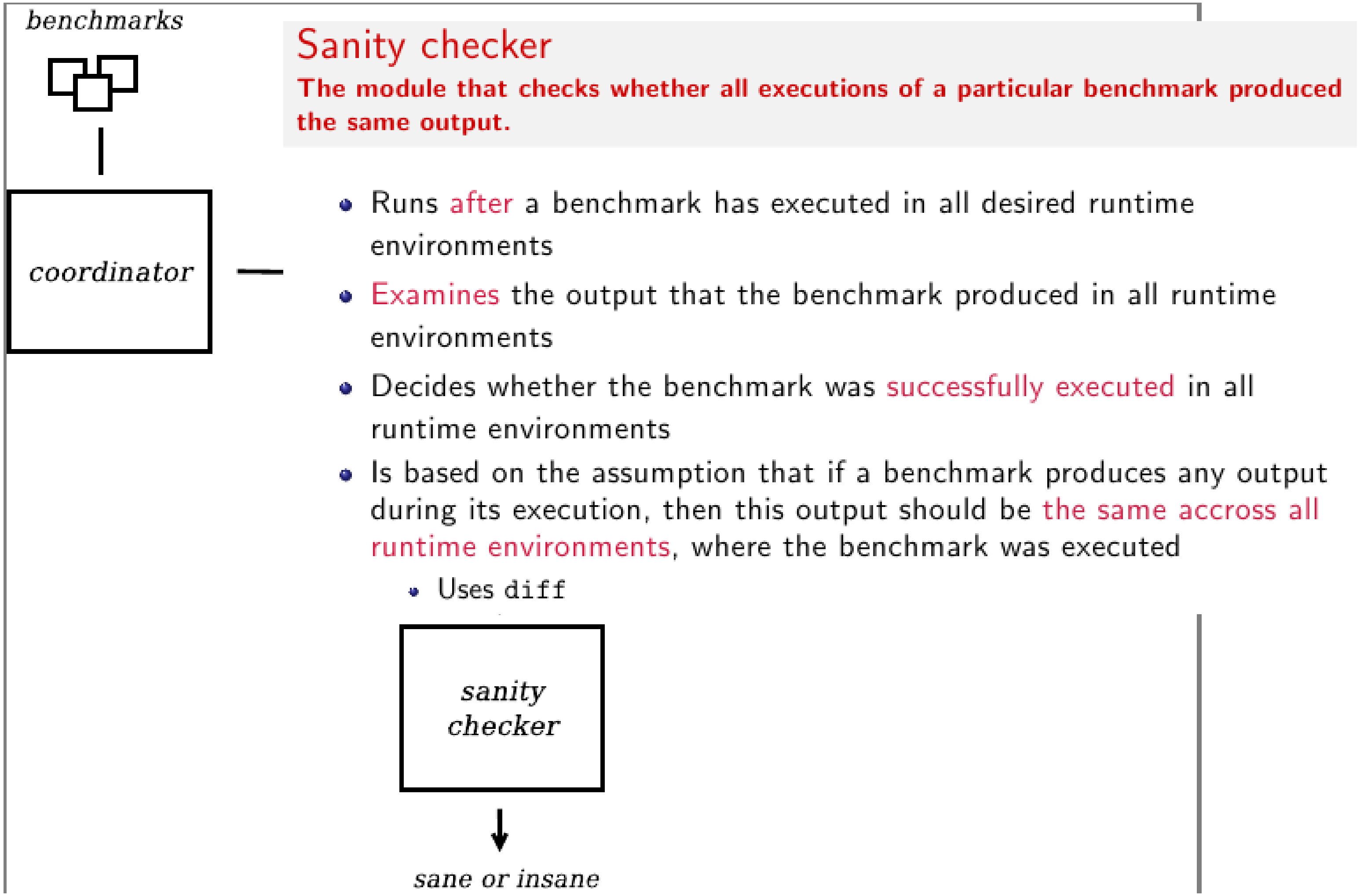
Executor

The module that executes a particular benchmark in a particular runtime environment.

- Receives detailed instructions from the executor about what to do
- Starts any necessary Erlang slave nodes
- Executes the benchmark in a new process
- Stops the Erlang slave nodes it started
- Makes sure that the output that the benchmark produced during its execution is written in an output file
- Makes sure that the measurements that are collected during the execution of the benchmark are written in a measument file
  - Uses `erlang:now/0` and `timer:diff/2`

RELEASE

BenchErl's Architecture

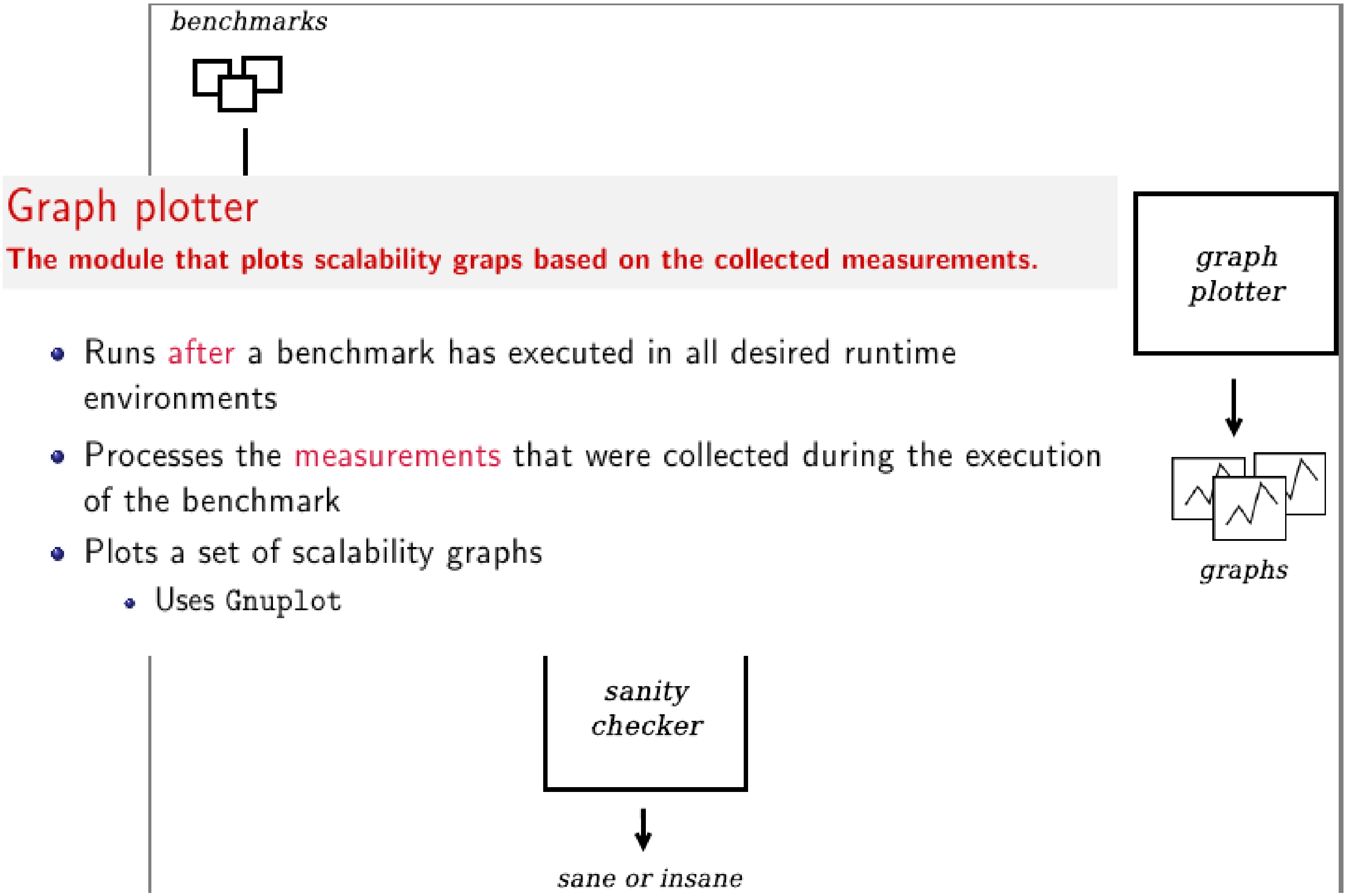


Sanity checker

The module that checks whether all executions of a particular benchmark produced the same output.

- Runs after a benchmark has executed in all desired runtime environments
- Examines the output that the benchmark produced in all runtime environments
- Decides whether the benchmark was successfully executed in all runtime environments
- Is based on the assumption that if a benchmark produces any output during its execution, then this output should be the same accross all runtime environments, where the benchmark was executed
  - Uses `diff`

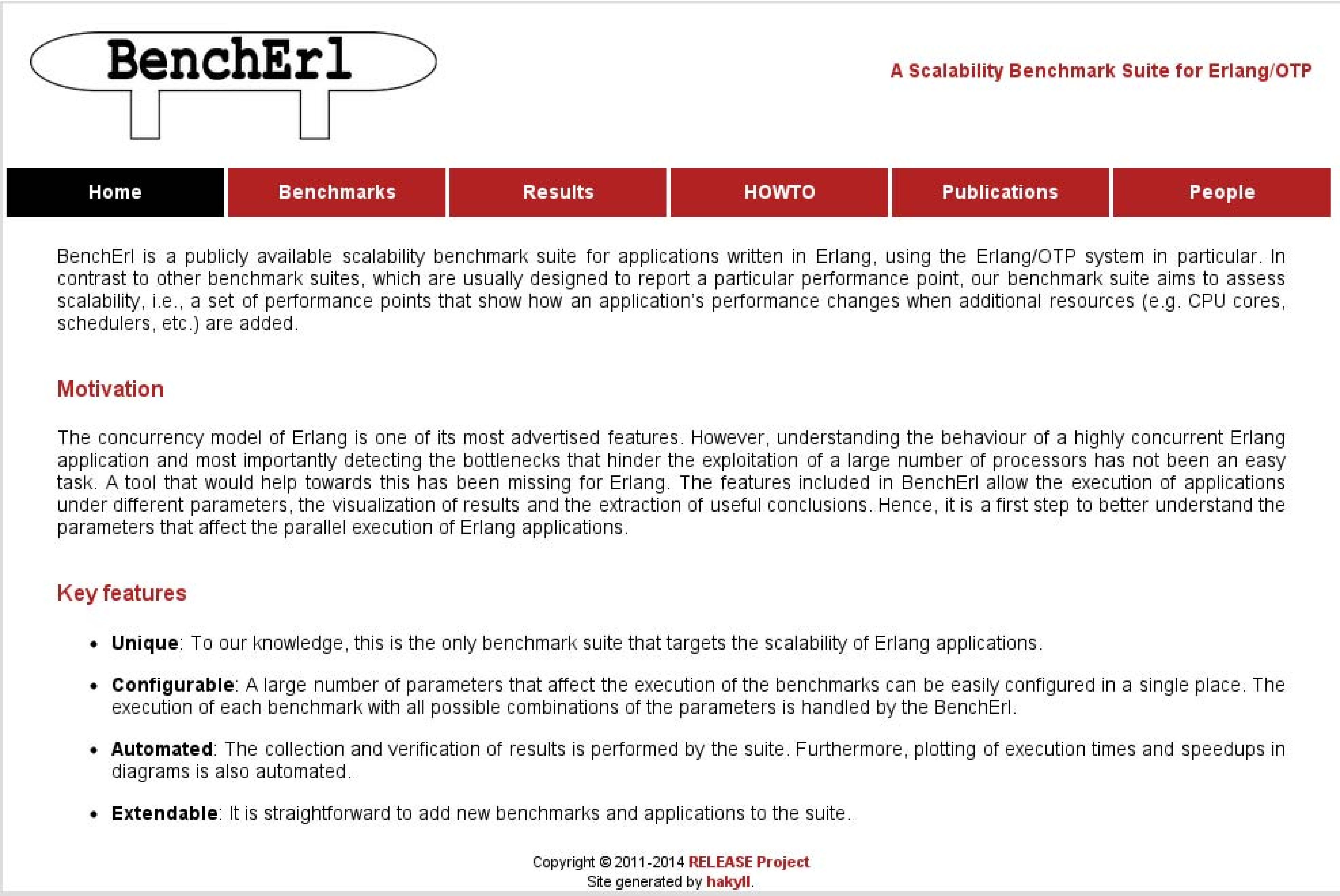
BenchErl's Architecture



Graph plotter

The module that plots scalability graps based on the collected measurements.

- Runs after a benchmark has executed in all desired runtime environments
- Processes the measurements that were collected during the execution of the benchmark
- Plots a set of scalability graphs
  - Uses `Gnuplot`



BenchErl

A Scalability Benchmark Suite for Erlang/OTP

BenchErl is a publicly available scalability benchmark suite for applications written in Erlang, using the Erlang/OTP system in particular. In contrast to other benchmark suites, which are usually designed to report a particular performance point, our benchmark suite aims to assess scalability, i.e., a set of performance points that show how an application's performance changes when additional resources (e.g. CPU cores, schedulers, etc.) are added.

Motivation

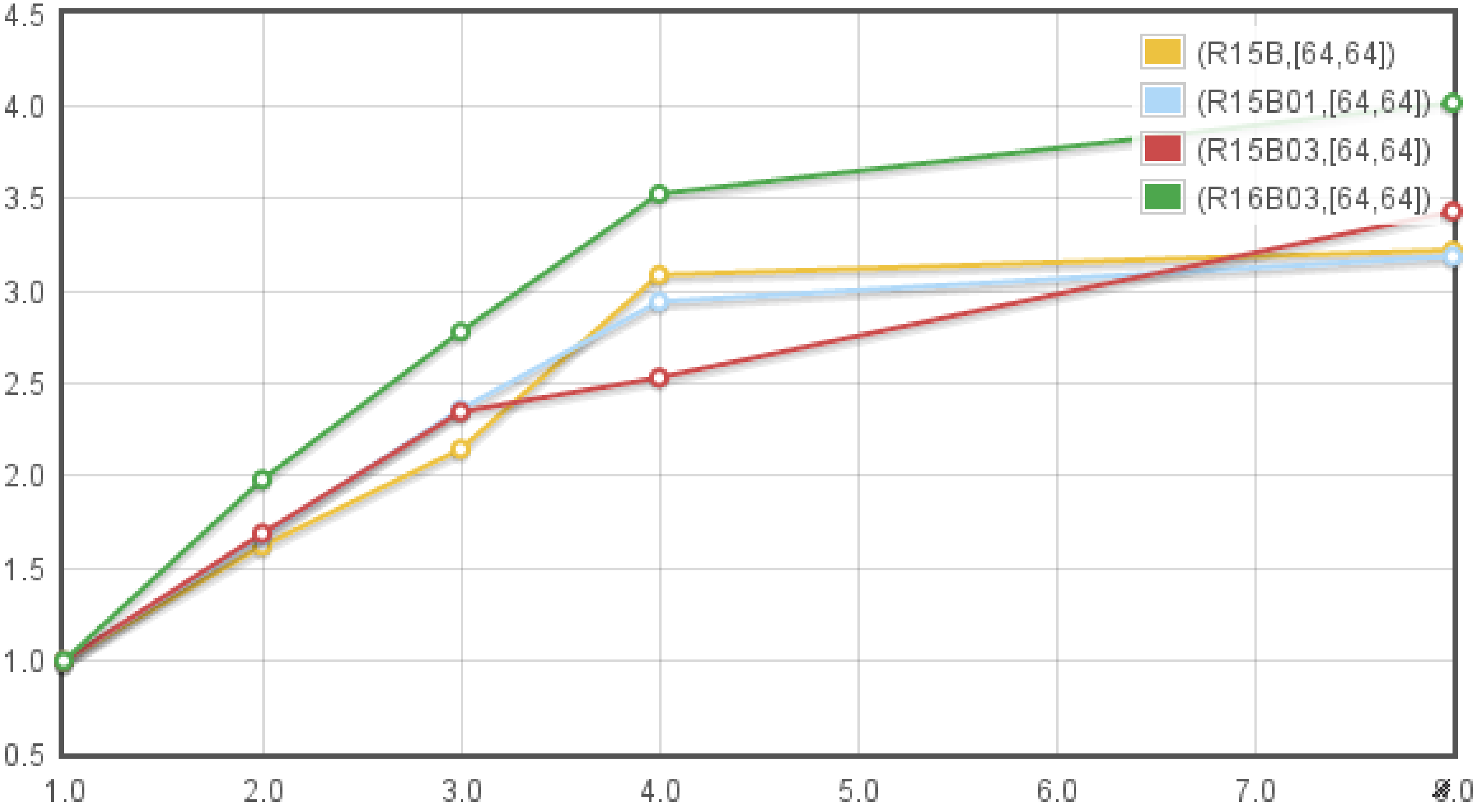
The concurrency model of Erlang is one of its most advertised features. However, understanding the behaviour of a highly concurrent Erlang application and most importantly detecting the bottlenecks that hinder the exploitation of a large number of processors has not been an easy task. A tool that would help towards this has been missing for Erlang. The features included in BenchErl allow the execution of applications under different parameters, the visualization of results and the extraction of useful conclusions. Hence, it is a first step to better understand the parameters that affect the parallel execution of Erlang applications.

Key features

- **Unique**: To our knowledge, this is the only benchmark suite that targets the scalability of Erlang applications.
- **Configurable**: A large number of parameters that affect the execution of the benchmarks can be easily configured in a single place. The execution of each benchmark with all possible combinations of the parameters is handled by the BenchErl.
- **Automated**: The collection and verification of results is performed by the suite. Furthermore, plotting of execution times and speedups in diagrams is also automated.
- **Extendable**: It is straightforward to add new benchmarks and applications to the suite.

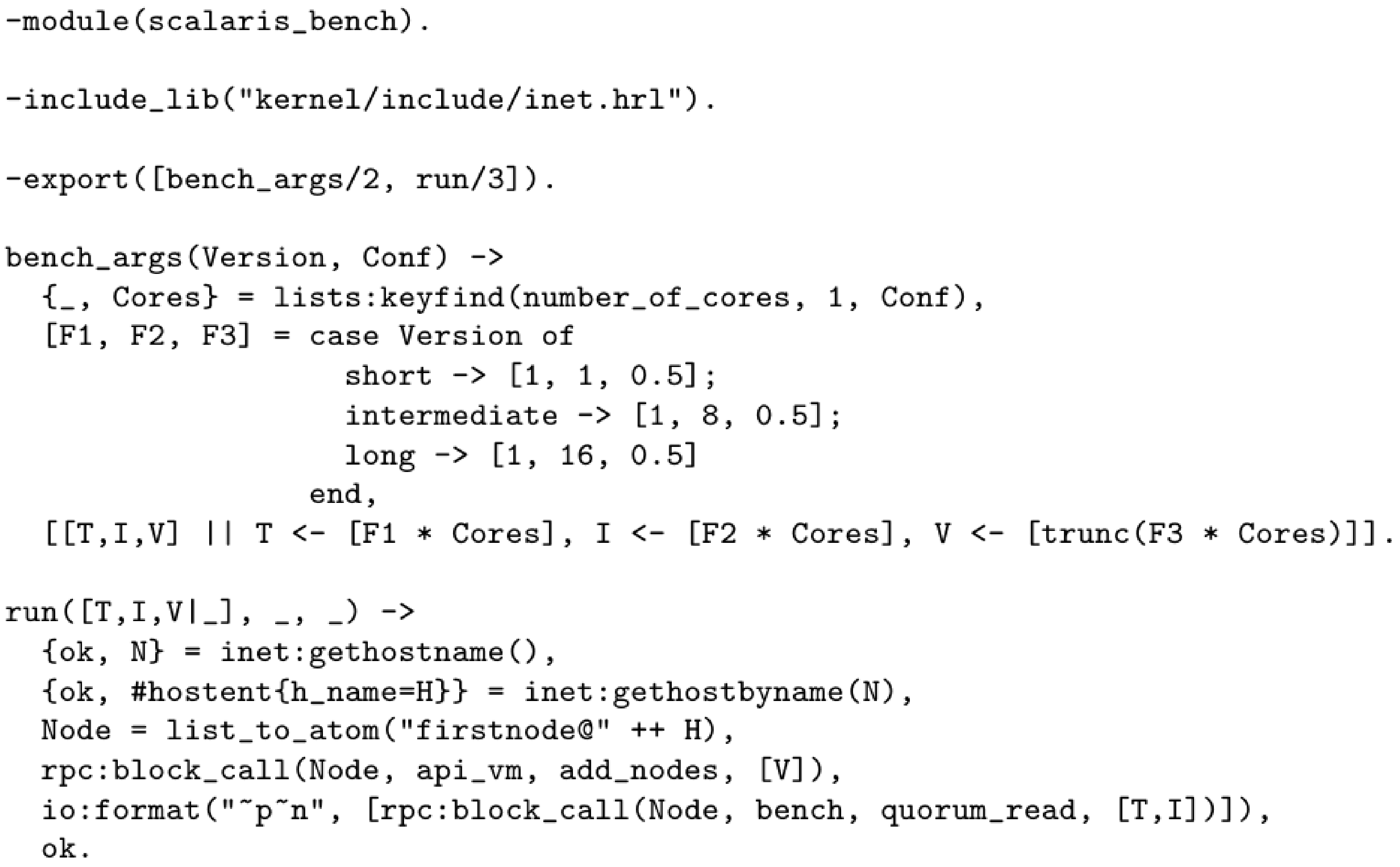
Site generated by hakyll.

ehb (“erlang-hack-bench”) on i7



RELEASE

Extending the Benchmark Suite



```
-module(scalaris_bench).

-include_lib("kernel/include/inet.hrl").

-export([bench_args/2, run/3]).

bench_args(Version, Conf) ->
  {_, Cores} = lists:keyfind(number_of_cores, 1, Conf),
  [F1, F2, F3] = case Version of
                   short -> [1, 1, 0.5];
                   intermediate -> [1, 8, 0.5];
                   long -> [1, 16, 0.5]
                 end,
  [[T,I,V] || T <- [F1 * Cores], I <- [F2 * Cores], V <- [trunc(F3 * Cores)]].

run([T,I,V|_], _, _) ->
  {ok, N} = inet:gethostname(),
  {ok, #hostent{h_name=H}} = inet:gethostbyname(N),
  Node = list_to_atom("firstnode@" ++ H),
  rpc:block_call(Node, api_vm, add_nodes, [V]),
  io:format("~p~n", [rpc:block_call(Node, bench, quorum_read, [T,I])]),
  ok.
```

On the Scalability of the Erlang Term Storage

David Klaftenegger

Kjell Winblad

Kostis Sagonas

RELEASE

UPMARC

UPPSALA UNIVERSITET

Erlang Term Storage (ETS)

- Key component of Erlang/OTP
  - Key/value store mechanism
  - Heavily used in applications
  - Supports **`mnesia`**
- Provides shared memory
  - with destructive updates!
  - crucial for parallelization
  - a scalability bottleneck?

Programming ETS

```
...
T = ets:new(mytable,
            [set, %bag, duplicate_bag, ordered_set
             public, %protected, private
             {keypos, 1},
             {read_concurrency, true},
             {write_concurrency, true}]),
ets:insert(T, [{key1,42}, {key2,val}]),
[{key1, V}] = ets:lookup(T, key1),
...
```

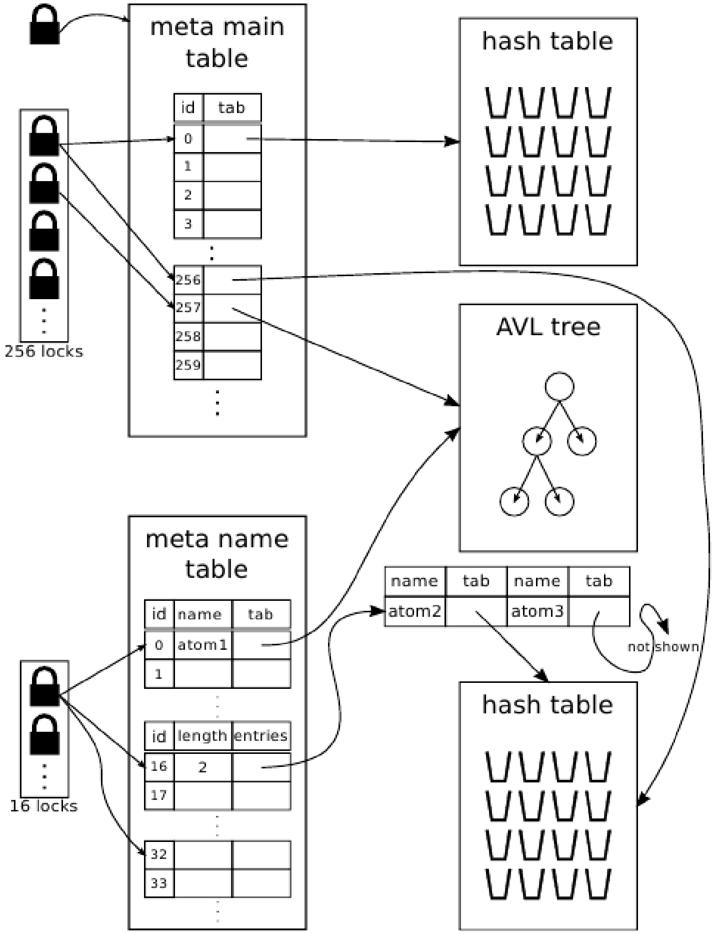
Implementation of ETS

Four types/two implementations

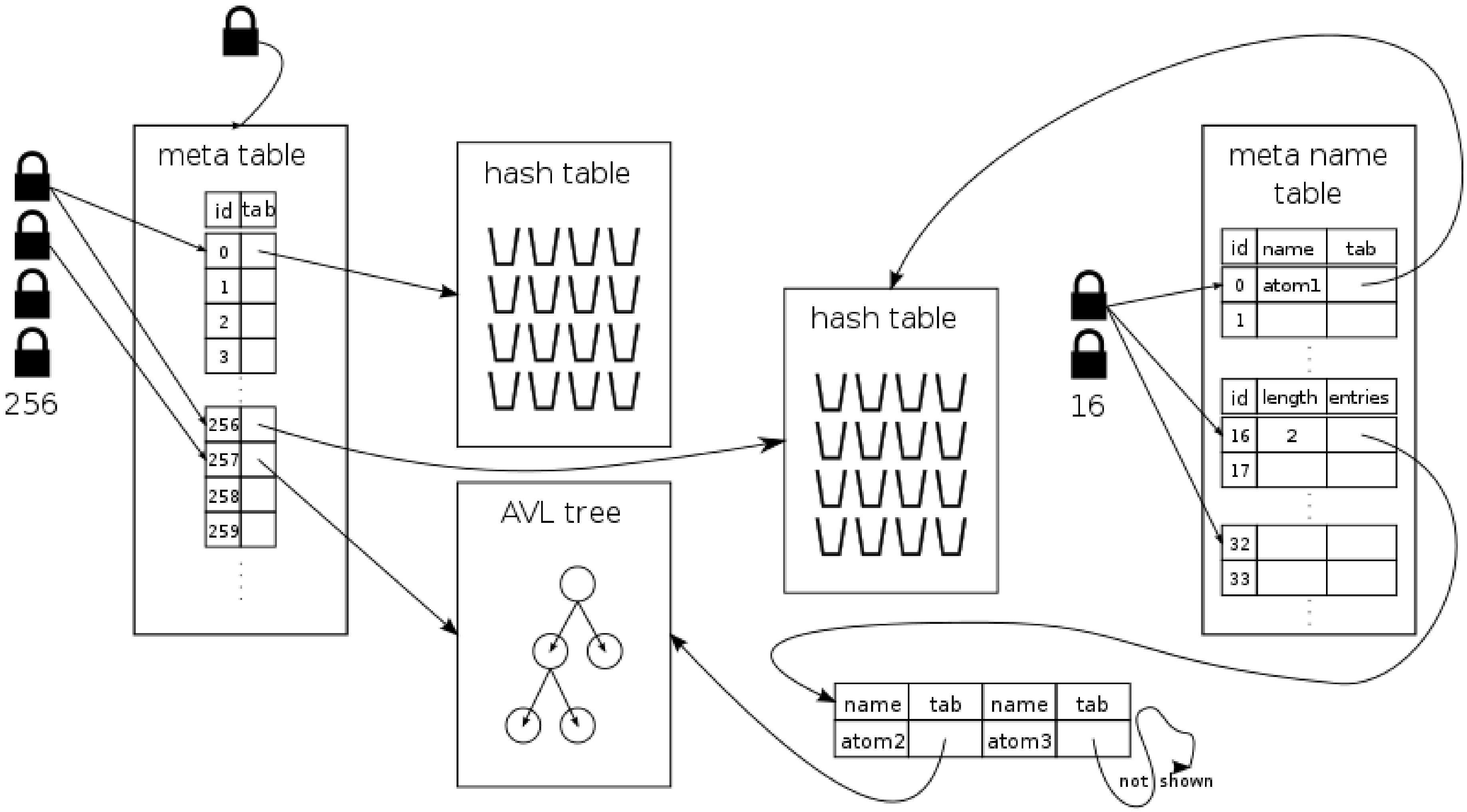
- **set**, **bag**, **duplicate_bag**
  - Linear Hash Tables
- **ordered_set**
  - AVL Trees

Concurrency options

- **write_concurrency**
- **read_concurrency**
- reader groups (**+rg**)
  - fine-grained locks



ETS Under the Hood



Linear Hash Tables

Hash key to bucket: bucket list

Resizing one bucket at a time

- Avg. bucket length: 6 in R16B

Locking

One readers-writer table lock

Bucket locks allow for fine-grained locking

Some operations need to lock the whole table

- Ex. insert all elements in a list atomically

AVL Trees

Used for ETS tables of type **`ordered_set`**

Balanced binary search trees

Locking

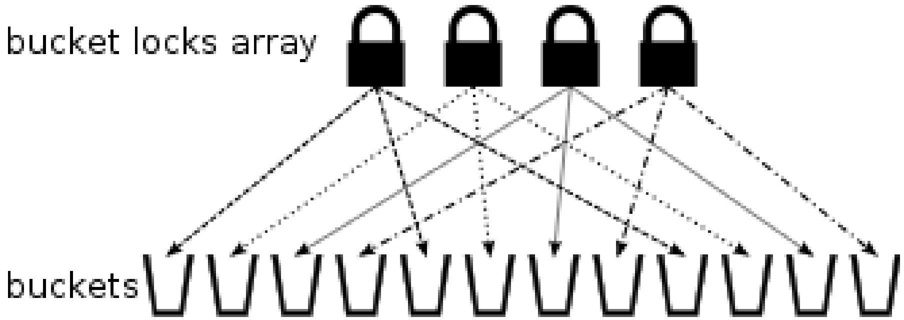
Protected by *single* readers-writer lock

Evolution of ETS

R11B → first version with SMP support

R13B02-1 → **`write_concurrency`**

Activates array of readers-writer locks



R13B02-1	16 locks
R16B	64 locks

R14B → **`read_concurrency`**

Schedulers are mapped to reader groups

Every reader group has its own read counter

R16B → more reader groups & more locks

ETS Benchmark

Table initialized with ~1M inserts (not measured)

Measure time to perform “random” ETS operations

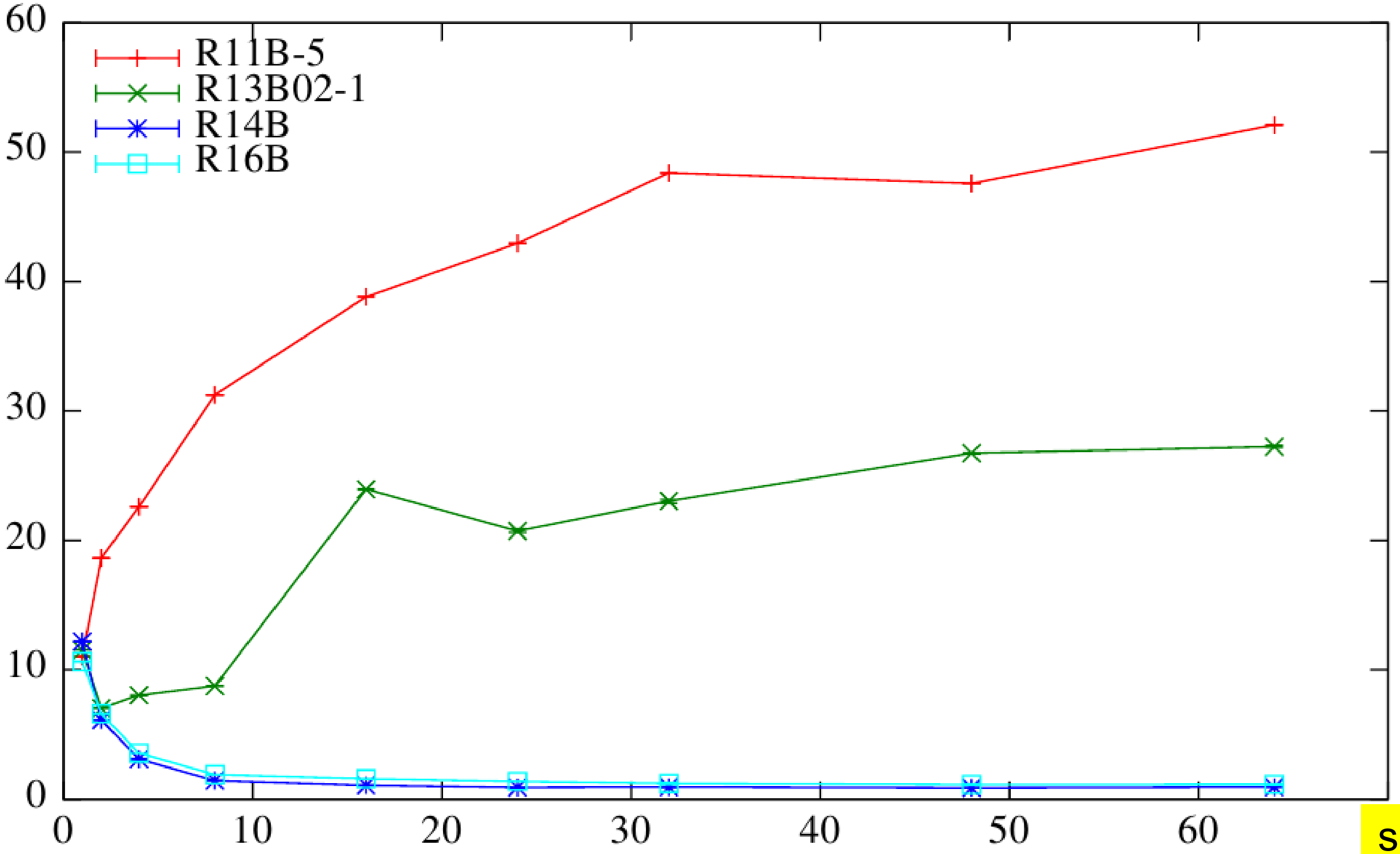
- Varying the percentage of lookups and updates
  - 90%, 99% and 100% lookups
  - updates are inserts & deletes
- Equal probability for inserts and deletes
  - size of table stays approximately the same

Machine with 32 Intel cores (64 with hyper-threading)

- Organized in four NUMA nodes (8 physical cores each)
- Schedulers are pinned to (logical) cores
  - One NUMA node at a time, filling the physical cores first

Scalability of ETS Across Releases

Time (secs)



schedulers

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

RELEASE

Scalability of ETS Across Releases

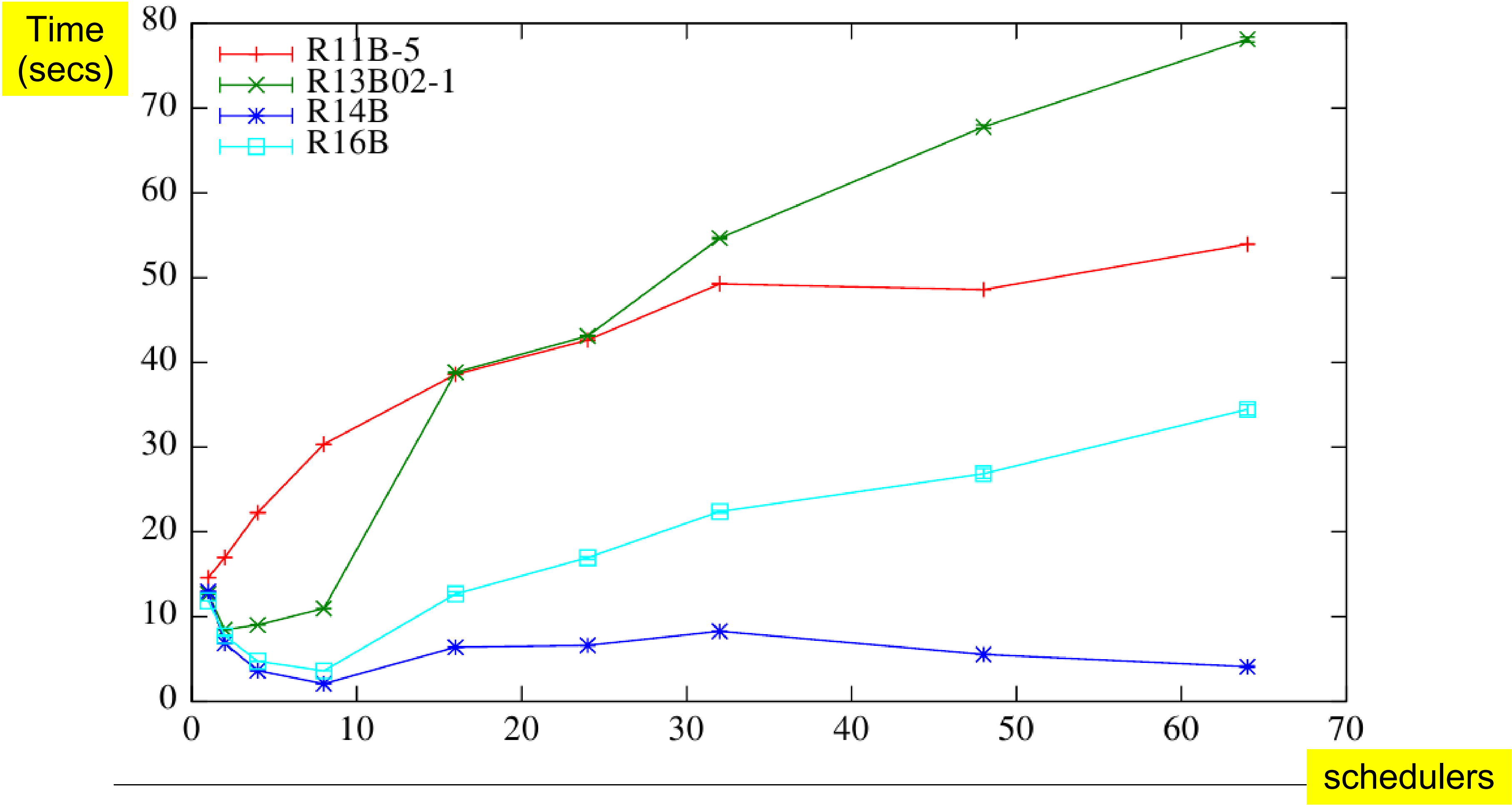


Figure 7. Scalability of ETS tables of type `ordered_set` across OTP releases using a workload with 99% lookups and 1% updates.

RELEASE

Effect of Concurrency Options (R16B)

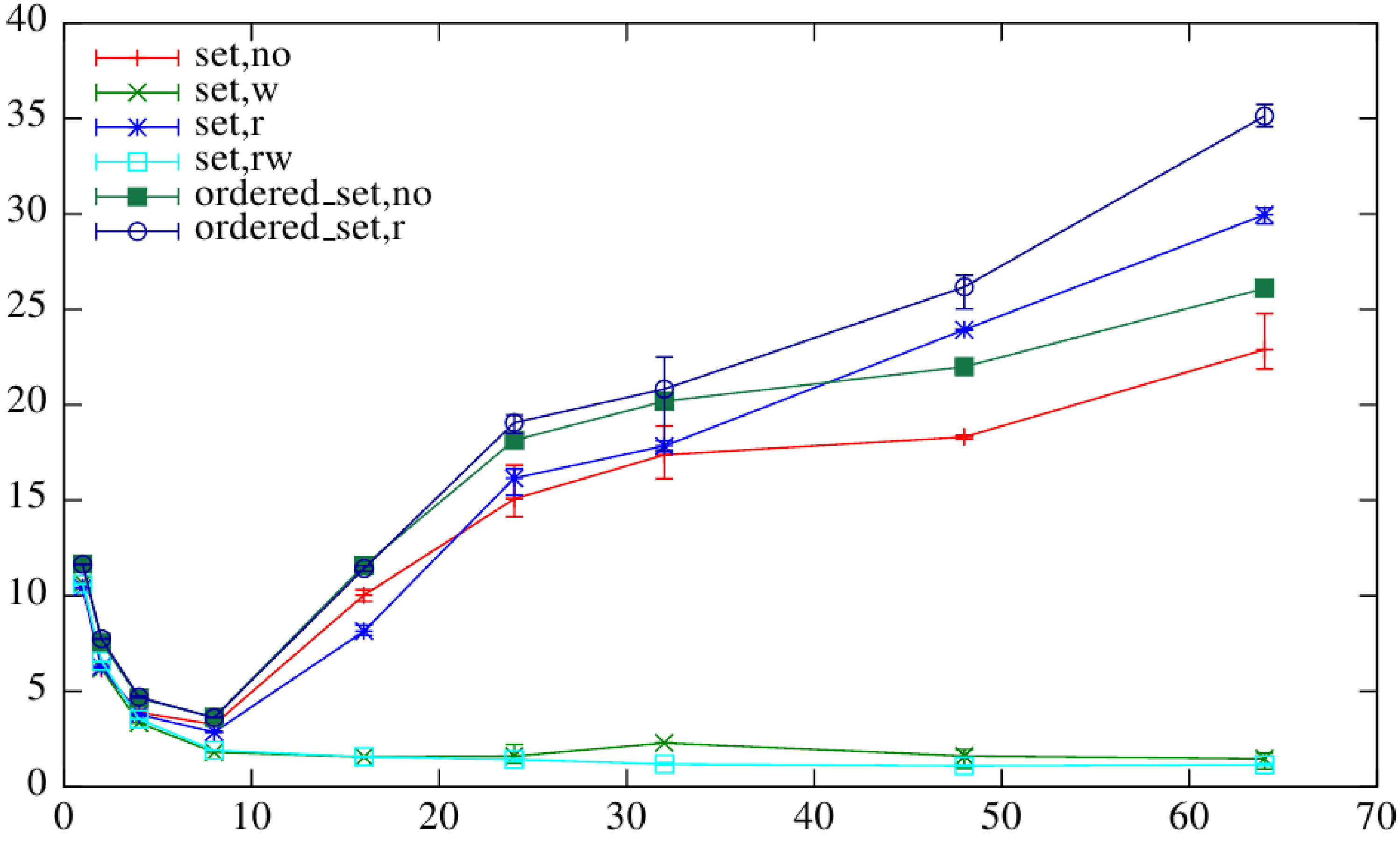


Figure 11. Scalability of ETS on a workload with 99% lookups and 1% updates when varying the ETS table concurrency options.

RELEASE

Effect of Reader Groups (R16B)

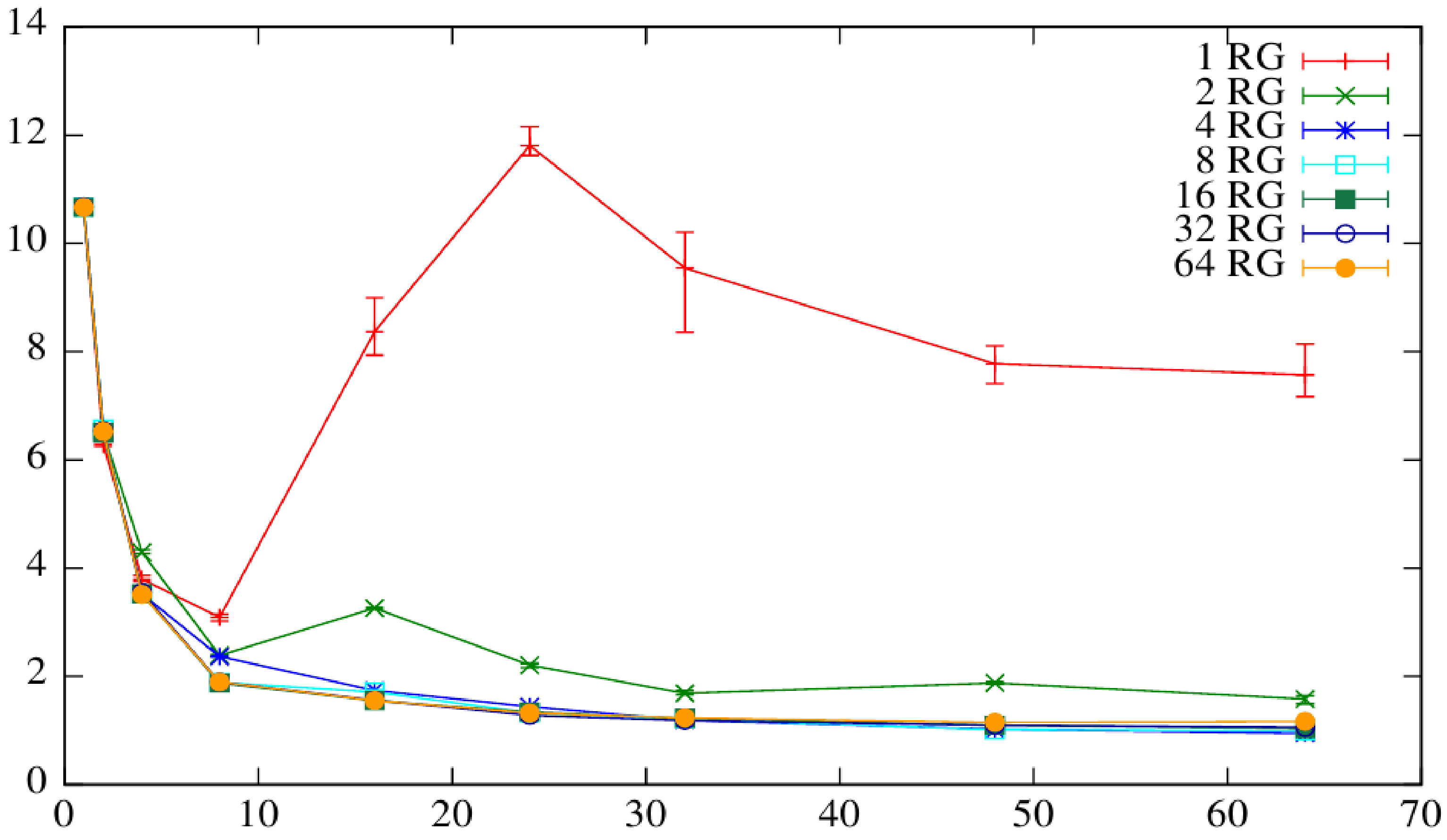


Figure 14. Scalability of ETS tables of type `set`, on 99% lookups and 1% updates, when varying the number of reader groups.

RELEASE

Effect of Reader Groups (R16B)

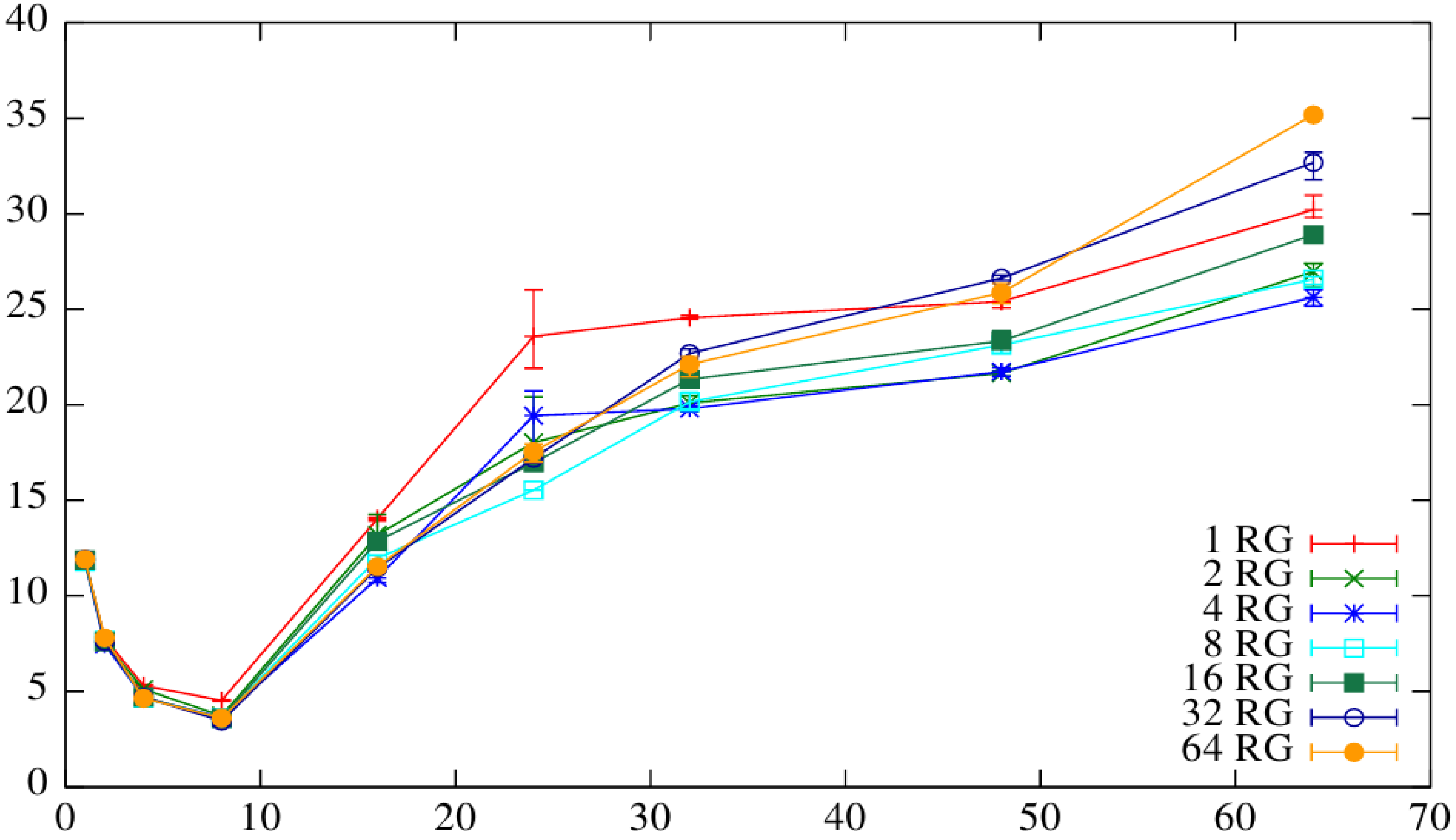


Figure 17. Scalability of tables of type `ordered_set`, on 99% lookups and 1% updates, varying the number of reader groups.

A Bottleneck in the AVL Tree

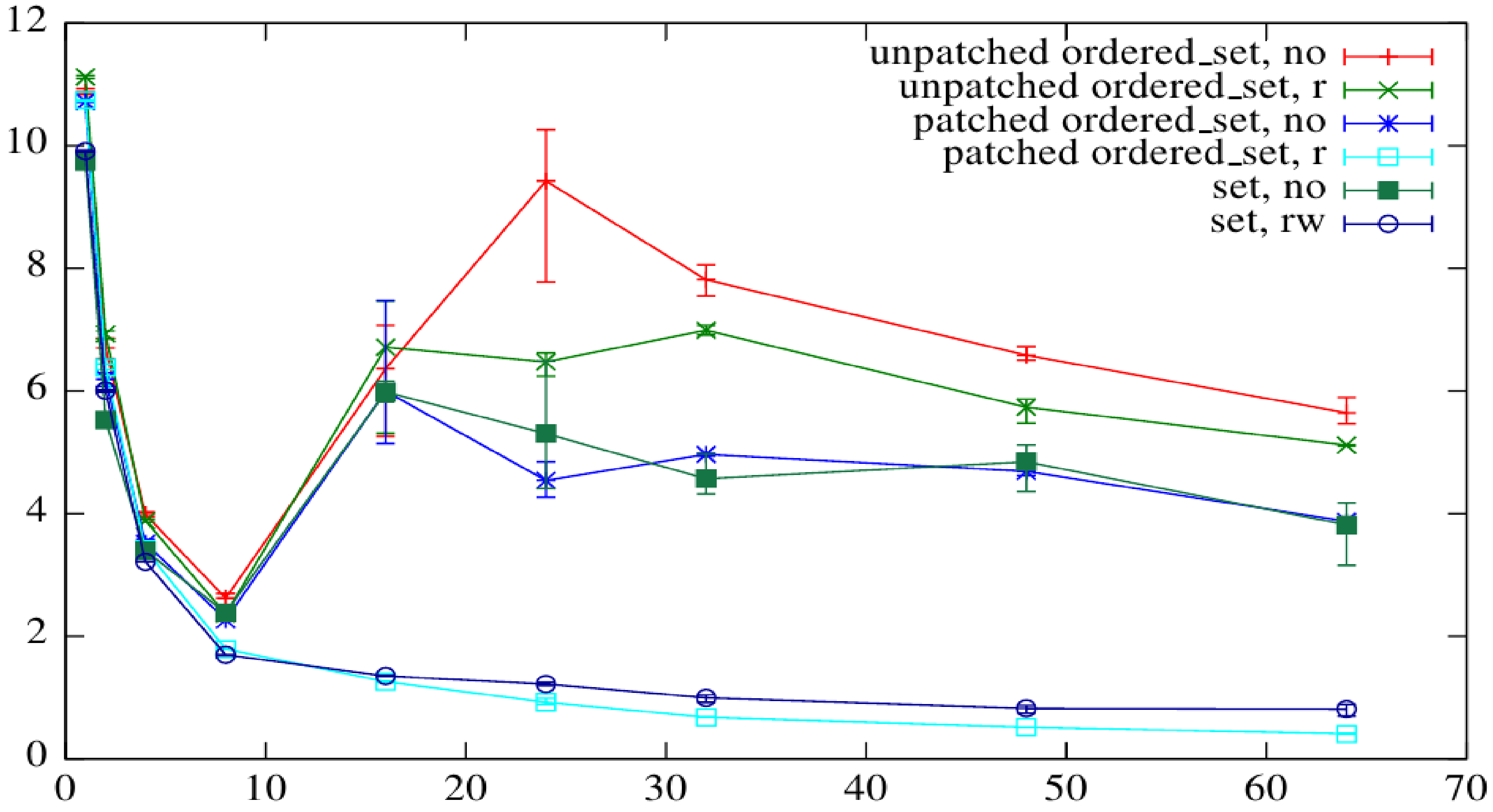


Figure 19. Scalability on a lookup-only workload when the AVL tree implementation uses a static (unpatched) vs. dynamic (patched) stack. (See Table 1 for the explanation of no, r and rw.)

More Scalable ordered_set Tables?

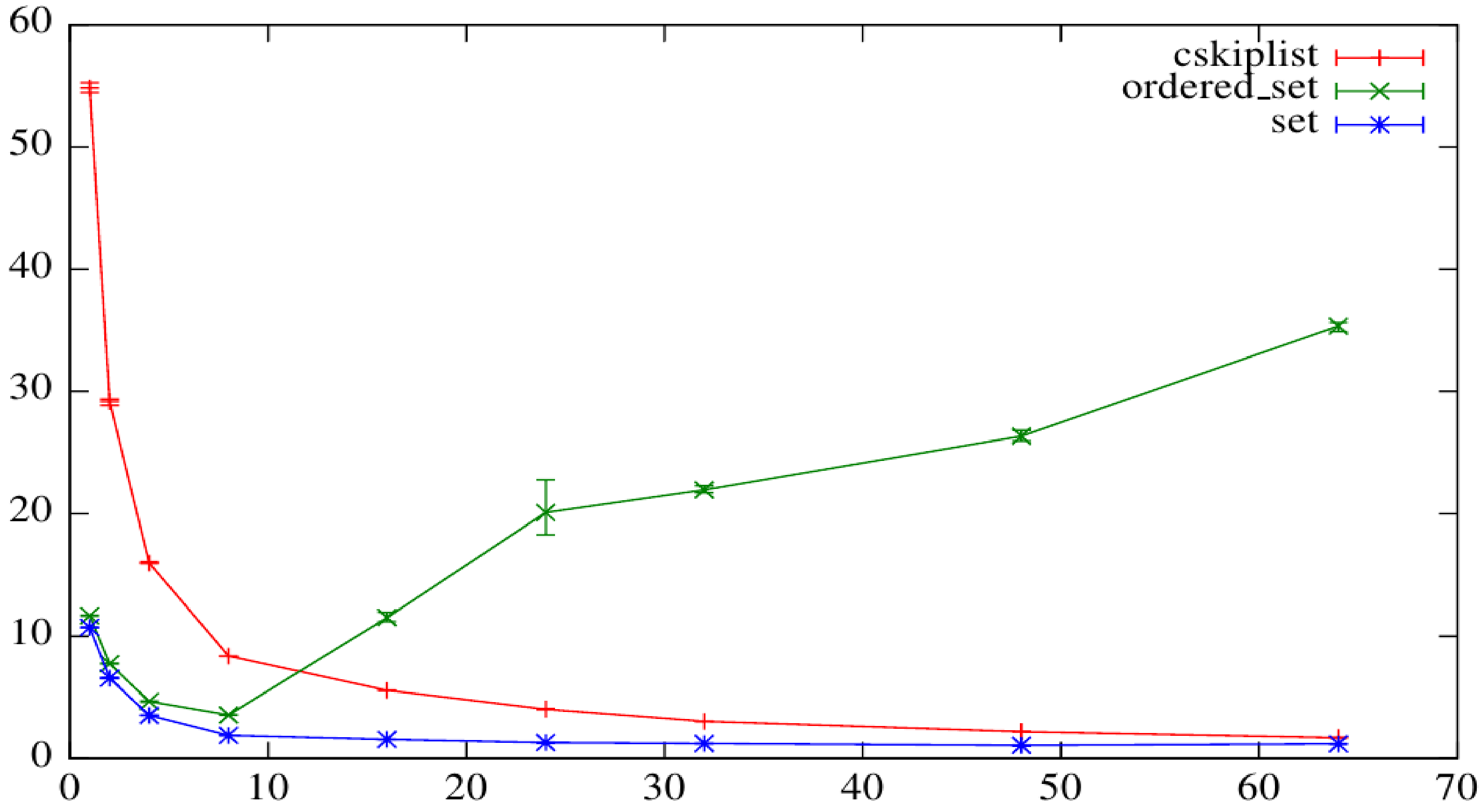


Figure 24. Comparison of a concurrent skip list against `set` and `ordered_set` on a workload with 99% lookups and 1% updates.

Scaling ETS: Lessons learned

- **`ordered_set`** needs to be fixed or replaced
- Locking is (still) a problem, but got better
- NUMA is a problem
- Reader groups may be not that important

Some general advice

- Use pinning on NUMA
- Use **`read_concurrency`** when doing only lookups
- Use **`write_concurrency`**
- Measure your use case when combining them

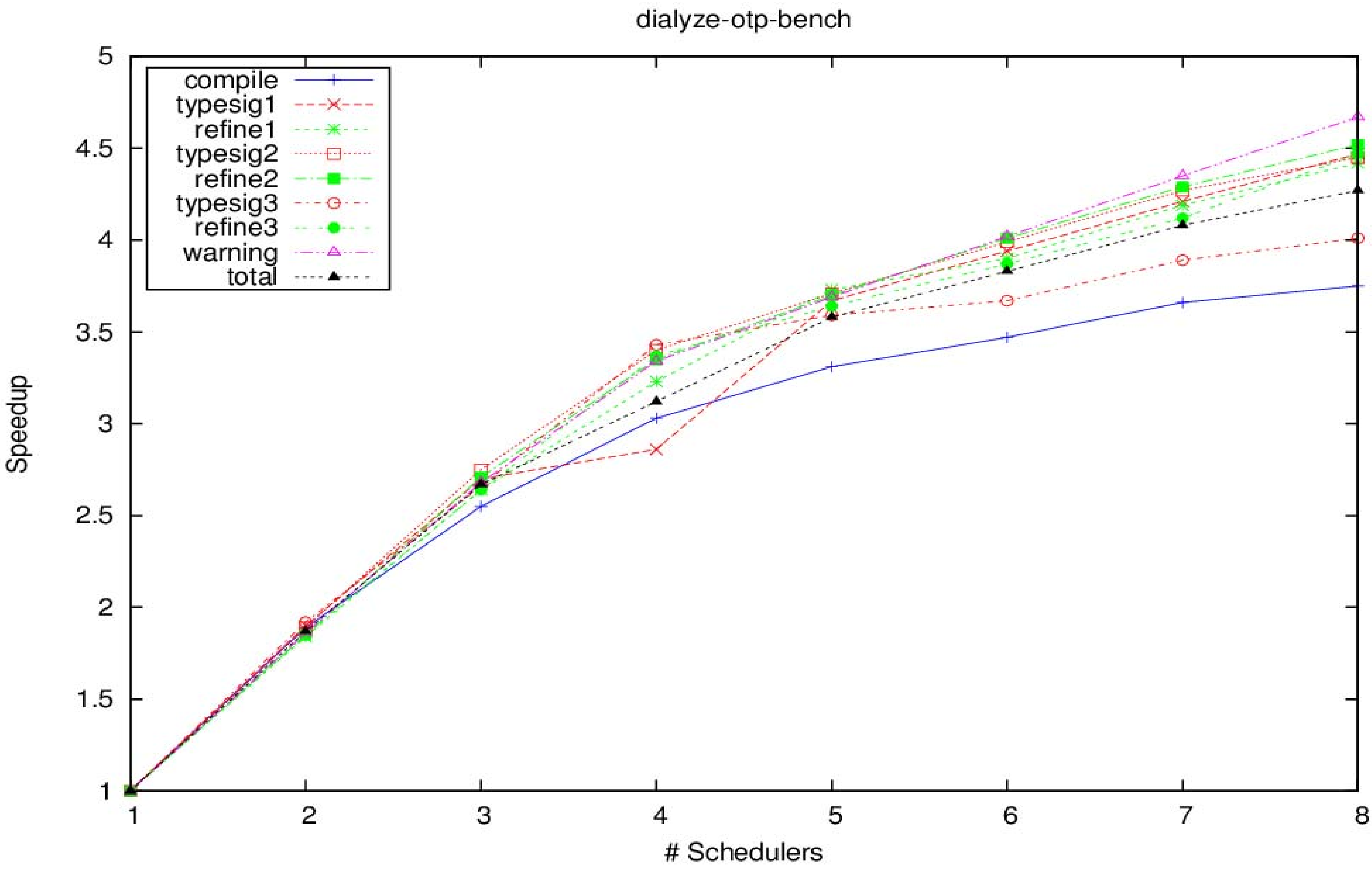
Thanks for your attention!

Parallelizing a Big Application

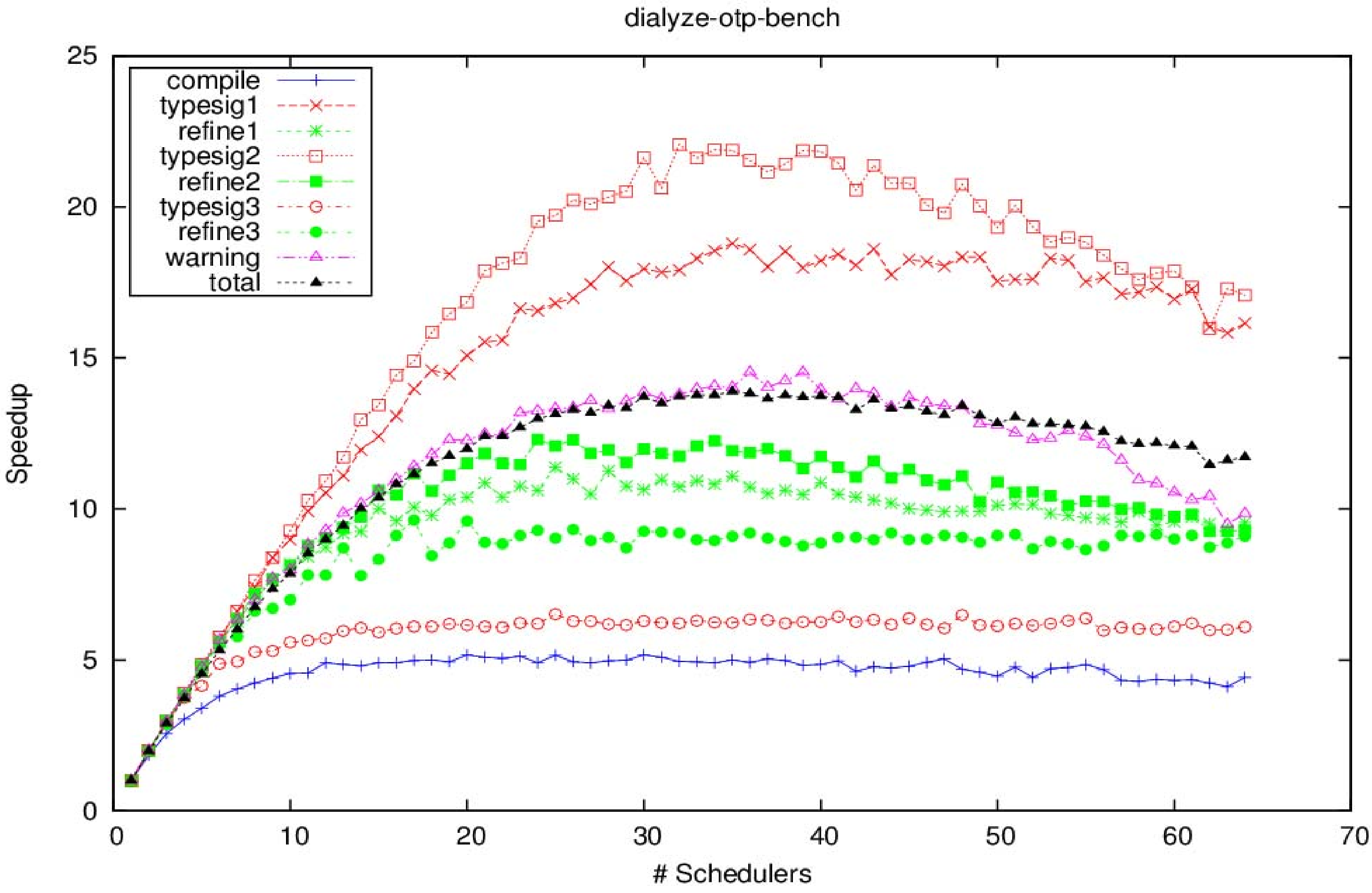
Dialyzer:

- Static analysis tool to find bugs in programs
- Developed over more than eight years now
- Heavily optimized & used
- About 30,000 lines of Erlang code
  - Memory hungry
  - Many phases – many synchronization points

Performance of Parallel Dialyzer



Performance of Parallel Dialyzer



RELEASE

More Information

- 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 11th ACM SIGPLAN Erlang Workshop, pp. 33-42, September 2012. ACM Press.
- D. Klaftenegger, K. Sagonas, and K. Winblad. On the Scalability of the Erlang Term Storage. In Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang, pp. 15-26, September 2013. ACM Press.
- S. Aronis and K. Sagonas. On using Erlang for parallelization: Experience from parallelizing Dialyzer. In Trends in Functional Programming, 13th International Symposium, pp. 295-310, Volume 7829 in LNCS, June 2012. Springer.