Scaling Reliably: Improving the Scalability of the Erlang Distributed Actor Platform

PHIL TRINDER, University of Glasgow
NATALIA CHECHINA, University of Glasgow
NIKOLAOS PAPASPYROU, National Technical University of Athens
KONSTANTINOS SAGONAS, Uppsala University and National Technical University of Athens
SIMON THOMPSON, University of Kent
STEPHEN ADAMS, University of Kent
STAVROS ARONIS, Uppsala University
ROBERT BAKER, University of Kent
EVA BIHARI, Erlang Solutions
OLIVIER BOUDEVILLE, EDF
FRANCESCO CESARINI, Erlang Solutions
MAURIZIO DI STEFANO, University of Kent
SVERKER ERIKSSON, Ericsson AB
VIKTÓRIA FÖRDŐS, Erlang Solutions
AMIR GHAFFARI, University of Glasgow
AGGELOS GIANTSIOS, National Technical University of Athens
RICKARD GREEN, Ericsson AB
CSABA HOCH, Erlang Solutions
DAVID KLAFTENEGGER, Uppsala University
HUIQING LI, University of Kent
KENNETH LUNDIN, Ericsson AB
KENNETH MACKENZIE, University of Glasgow
KATERINA ROUKOUNAKI, National Technical University of Athens
YIANNIS TSIOURIS, National Technical University of Athens
KJELL WINBLAD, Uppsala University

This work has been supported by the European Union grants RII3-CT-2005-026133 and IST-2011-287510, and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1.
Author's addresses: P.W. Trinder, School of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland.

DOI: 0000001.0000001

Distributed actor languages are an effective means of constructing scalable reliable systems, and the Erlang programming language has a well-established and influential model. While the Erlang model conceptually provides reliable scalability, it has some inherent scalability limits and these force developers to depart from the model at scale. This article establishes the scalability limits of Erlang systems, and reports the work of the EU RELEASE project to improve the scalability and understandability of the Erlang reliable distributed actor model.

We systematically study the scalability limits of Erlang, and then address the issues at the virtual machine, language and tool levels. More specifically: (1) We have evolved the Erlang virtual machine so that it can work effectively in large scale single-host multicore and NUMA architectures. We have made important changes and architectural improvements to the widely used Erlang/OTP release. (2) We have designed and implemented Scalable Distributed (SD) Erlang libraries to address language-level scalability issues, and provided and validated a set of semantics for the new language constructs. (3) To make large Erlang systems easier to deploy, monitor, and debug we have developed and made open source releases of five complementary tools, some specific to SD Erlang.

Throughout the article we use two case studies to investigate the capabilities of our new technologies and tools: a distributed hash table based Orbit calculation and Ant Colony Optimisation (ACO). Chaos Monkey experiments show that two versions of ACO survive random process failure and hence that SD Erlang preserves the Erlang reliability model. While we report measurements on a range of NUMA and cluster architectures, the key scalability experiments are conducted on the Athos cluster with 256 hosts (6144 cores). Even for programs with no global recovery data to maintain, SD Erlang partitions the network to reduce network traffic and hence improves performance of the Orbit and ACO benchmarks above 80 hosts. ACO measurements show that maintaining global recovery data dramatically limits scalability; however scalability is recovered by partitioning the recovery data. We exceed the established scalability limits of distributed Erlang, and do not reach the limits of SD Erlang for these benchmarks at this scale (256 hosts, 6144 cores).

CCS Concepts: •**Software and its engineering** → **Software fault tolerance; Distributed programming languages;** *Functional languages;*

Additional Key Words and Phrases: Erlang, scalability, reliability

ACM Reference Format:
Phil Trinder, Natalia Chechina, Nikolaos Papaspyrou, Konstantinos Sagonas, Simon Thompson, *et al.* 2017. Scaling Reliably: Improving the Scalability of the Erlang Distributed Actor Platform. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (2017), 46 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Distributed programming languages and frameworks are central to engineering large scale systems, where key properties include scalability and reliability. By scalability we mean that performance increases as hosts and cores are added, and by large scale we mean architectures with hundreds of hosts and tens of thousands of cores. Experience with high performance and data centre computing shows that reliability is critical at these scales, e.g. host failures alone account for around one failure per hour on commodity servers with approximately 10^5 cores [Barroso et al. 2013]. To be usable, programming languages employed on them must be supported by a suite of deployment, monitoring, refactoring and testing tools that work at scale.

Controlling shared state is the only way to build reliable scalable systems. State shared by multiple units of computation limits scalability due to high synchronisation and communication costs. Moreover shared state is a threat for reliability as failures corrupting or permanently locking shared state may poison the entire system.

Actor languages avoid shared state: actors or processes have entirely local state, and only interact with each other by sending messages [Agha 1986]. Recovery is facilitated in this model, since actors, like operating system processes, can fail independently without affecting the state of other actors. Moreover an actor can *supervise* other actors, detecting failures and taking remedial action, e.g. restarting the failed actor.

Erlang [Armstrong 2007; Cesarini and Thompson 2009] is a beacon language for reliable scalable computing with a widely emulated distributed actor model. It has influenced the design of numerous programming languages like Clojure [Hickey 2008] and F# [Syme et al. 2015], and many languages have Erlang-inspired actor frameworks, e.g. Kilim for Java [Srinivasan and Mycroft 2008], Cloud Haskell [Epstein et al. 2011], and Akka for C#, F# and Scala [Odersky et al. 2012]. Erlang is widely used for building reliable scalable servers, e.g. Ericsson's AXD301 telephone exchange (switch) [Wiger 2000], the Facebook chat server, and the Whatsapp instant messaging server [WhatsApp 2015].

In Erlang, the actors are termed processes and are managed by a sophisticated Virtual Machine on a single multicore or NUMA host, while *distributed Erlang* provides relatively transparent distribution over networks of VMs on multiple hosts. Erlang is supported by the Open Telecom Platform (OTP) libraries that capture common patterns of reliable distributed computation, such as the client-server pattern and process supervision. Any large-scale system needs scalable persistent storage and, following the CAP theorem [Gilbert and Lynch 2002], Erlang uses and indeed implements Dynamo-style NoSQL DBMS like Riak [Klophaus 2010] and Cassandra [Lakshman and Malik 2010].

While the Erlang distributed actor model conceptually provides reliable scalability, it has some inherent scalability limits, and indeed large-scale distributed Erlang systems must depart from the distributed Erlang paradigm in order to scale, e.g. not maintaining a fully connected graph of hosts. The EU FP7 RELEASE project set out to establish and address the scalability limits of the Erlang reliable distributed actor model [RELEASE Project Team 2015].

After outlining related work (Section 2) and the benchmarks used throughout the article (Section 3) we investigate the scalability limits of Erlang/OTP, seeking to identify specific issues at the virtual machine, language and persistent storage levels (Section 4). We then report the RELEASE project work to address these issues, working at the following three levels.

(1) We have designed and implemented a set of Scalable Distributed (SD) Erlang libraries to address *language-level* reliability and scalability issues. An operational semantics is provided for the key new s_group construct, and the implementation is validated against the semantics (Section 5).
(2) We have *evolved the Erlang virtual machine* so that it can work effectively in large-scale single-host multicore and NUMA architectures. We have improved the shared ETS tables, time management, and load balancing between schedulers. Most of these improvements are now included in the Erlang/OTP release, currently downloaded approximately 50K times each month (Section 6).
(3) To facilitate the development of scalable Erlang systems, and to make them maintainable, we have developed three new tools: Devo, SDMon and WombatOAM, and enhanced two others: the visualisation tool Percept, and the refactorer Wrangler. The tools support refactoring programs to make them more scalable, easier to deploy at large scale (hundreds of hosts), easier to monitor and visualise their behaviour. Most of these tools are freely available under open source licences; the WombatOAM deployment and monitoring tool is a commercial product (Section 7).

Throughout the article we use two benchmarks to investigate the capabilities of our new technologies and tools. These are a computation in symbolic algebra, more specifically an algebraic 'orbit' calculation that exploits a non-replicated distributed hash table, and an Ant Colony Optimisation (ACO) parallel search program (Section 3).

We report on the reliability and scalability implications of our new technologies using Orbit to exhibit strong scaling, ACO to exhibit weak scaling, in addition to other

benchmark measurements. We use a Chaos Monkey instance [Bennett and Tseitlin 2012] that randomly kills processes in the running system to demonstrate the reliability of the benchmarks and to show that SD Erlang preserves the Erlang language-level reliability model. While we report measurements on a range of NUMA and cluster architectures as specified in Appendix A, the key scalability experiments are conducted on the Athos cluster with 256 hosts and 6144 cores. Having established scientifically the folklore limitations of around 60 connected hosts/nodes for distributed Erlang systems in Section 4, a key result is to show that the SD Erlang benchmarks exceed this limit and do not reach their limits on the Athos cluster (Section 8).

Contributions. This article is the first systematic presentation of the coherent set of technologies for engineering scalable reliable Erlang systems developed in the RELEASE project.

Section 4 presents the first scalability study covering Erlang VM, language, and storage scalability. Indeed we believe it is the first comprehensive study of any distributed actor language at this scale (100s of hosts, and around 10K cores). Individual scalability studies, e.g. into Erlang VM scaling [Aronis et al. 2012], or language and storage scaling have appeared before [Ghaffari et al. 2013; Ghaffari 2014b].

At the language level the design, implementation and validation of the new libraries (Section 5) have been reported piecemeal [Chechina et al. 2016; MacKenzie et al. 2015], and are included here for completeness.

While some of the improvements made to the Erlang Virtual Machine (Section 6.1) have been thoroughly reported in conference publications [Papaspyrou and Sagonas 2012; Klaftenegger et al. 2013; 2014; Sagonas and Winblad 2014; 2015; 2016], others are reported here for the first time (Sections 6.2 and 6.3).

In Section 7, the WombatOAM and SD-Mon tools are described for the first time, as is the revised Devo system and visualisation. The other tools for profiling, debugging and refactoring developed in the project have previously been published piecemeal [Li and Thompson 2012; 2013; 2015; Baker et al. 2013], but this is their first unified presentation.

All of the performance results in Section 8 are entirely new, although a comprehensive study of SD Erlang performance is now available in a recent article by Chechina et al. [2017].

2. CONTEXT

2.1. Scalable Reliable Programming Models

There is a plethora of shared memory concurrent programming models like PThreads or Java threads, and some models, like OpenMP [Chandra et al. 2001], are simple and high level. However synchronisation costs mean that these models generally do not scale well, often struggling to exploit even 100 cores. Moreover, reliability mechanisms are greatly hampered by the shared state: for example, a lock becomes permanently unavailable if the thread holding it fails.

The High Performance Computing (HPC) community build large-scale (10^6 core) distributed memory systems using the *de facto* standard MPI communication libraries [Snir et al. 1995]. Increasingly these are hybrid applications that combine MPI with OpenMP. Unfortunately, MPI is not suitable for producing general purpose concurrent software as it is too low level with explicit message passing. Moreover, the most widely used MPI implementations offer no fault recovery:[1] if any part of the computation fails, the entire computation fails. Currently the issue is addressed by using what is hoped to be highly reliable computational and networking hardware, but there

[1]Some fault tolerance is provided in less widely used MPI implementations like [Dewolfs et al. 2006].

is intense research interest in introducing reliability into HPC applications [Gainaru and Cappello 2015].

Server farms use commodity computational and networking hardware, and often scale to around 10^5 cores, where host failures are routine. They typically perform rather constrained computations, e.g. Big Data Analytics, using reliable frameworks like Google MapReduce [Dean and Ghemawat 2008] or Hadoop [White 2012]. The idempotent nature of the analytical queries makes it relatively easy for the frameworks to provide implicit reliability: queries are monitored and failed queries are simply re-run. In contrast, actor languages like Erlang are used to engineer reliable general purpose computation, often recovering failed stateful computations.

2.2. Actor Languages

The actor model of concurrency consists of independent processes communicating by means of messages sent asynchronously between processes. A process can send a message to any other process for which it has the address (in Erlang the "process identifier" or pid), and the remote process may reside on a different host. While the notion of actors originated in AI [Hewitt et al. 1973], it has been used widely as a general metaphor for concurrency, as well as being incorporated into a number of niche programming languages in the 1970s and 80s. More recently it has come back to prominence through the rise of not only multicore chips but also larger-scale distributed programming in data centres and the cloud.

With built-in concurrency and data isolation, actors are a natural paradigm for engineering reliable scalable general-purpose systems [Agha 1985; Hewitt 2010]. The model has two main concepts: actors that are the unit of computation, and messages that are the unit of communication. Each actor has an *address-book* that contains the addresses of all the other actors it is aware of. These addresses can be either locations in memory, direct physical attachments, or network addresses. In a pure actor language, messages are the only way for actors to communicate.

After receiving a message an actor can do the following: (i) send messages to another actor from its *address-book*, (ii) create new actors, or (iii) designate a behaviour to handle the next message it receives. The model does not impose any restrictions in the order in which these actions must be taken. Similarly, two messages sent concurrently can be received in any order. These features enable actor based systems to support indeterminacy and quasi-commutativity, while providing locality, modularity, reliability and scalability [Hewitt 2010].

Actors are just one message-based paradigm, and other languages and libraries have related message passing paradigms. Recent example languages include Go [Donovan and Kernighan 2015] and Rust [Matsakis and Klock II 2014] that provide explicit channels, similar to actor mailboxes. Probably the most famous message passing library is MPI [Snir et al. 1995], with APIs for many languages and widely used on clusters and High Performance Computers. It is, however, arguable that the most important contribution of the actor model is the one-way asynchronous communication [Hewitt 2010]. Messages are not coupled with the sender, and neither they are transferred synchronously to a temporary container where transmission takes place, e.g. a buffer, a queue, or a mailbox. Once a message is sent, the receiver is the only entity responsible for that message.

Erlang [Armstrong 2007; Cesarini and Thompson 2009] is the pre-eminent programming language based on the actor model, having a history of use in production systems, initially with its developer Ericsson and then more widely through open source adoption. There are now actor frameworks for many other languages; these include Akka

for C#, F# and Scala [Odersky et al. 2012], CAF[2] for C++, Pykka[3], Cloud Haskell [Epstein et al. 2011], PARLEY [Lee et al. 2010] for Python, and Termite Scheme [Germain 2006], and each of these is currently under active use and development. Moreover, the recently defined Rust language [Matsakis and Klock II 2014] has a version of the actor model built in, albeit in an imperative context.

2.3. Erlang's Support for Concurrency

In Erlang, actors are termed *processes*, and virtual machines are termed *nodes*. The key elements of the actor model are: fast process creation and destruction; lightweight processes, e.g. enabling 10^6 concurrent processes on a single host with 8GB RAM; fast asynchronous message passing with copying semantics; process monitoring; strong dynamic typing, and selective message reception.

By default Erlang processes are addressed by their *process identifier* (*pid*), e.g.

```
Pong_PID = spawn(fun some_module:pong/0)
```

spawns a process to execute the anonymous function given as argument to the `spawn` primitive, and binds `Pong_PID` to the new process identifier. Here the new process will execute the `pong/0` function which is defined in `some_module`. A subsequent call

```
Pong_PID ! finish
```

sends the messaged `finish` to the process identified by `Pong_PID`. Alternatively, processes can be given names using a call of the form:

```
register(my_funky_name, Pong_PID)
```

which registers this process name in the node's *process name table* if not already present. Subsequently, these names can be used to refer to or communicate with the corresponding processes (e.g. send them a message):

```
my_funky_process ! hello.
```

A *distributed Erlang* system executes on multiple nodes, and the nodes can be freely deployed across hosts, e.g. they can be located on the same or different hosts. To help make distribution transparent to the programmer, when any two nodes connect they do so transitively, sharing their sets of connections. Without considerable care this quickly leads to a fully connected graph of nodes. A process may be spawned on an explicitly identified node, e.g.

```
Remote_Pong_PID = spawn(some_node, fun some_module:pong/0).
```

After this, the remote process can be addressed just as if it were local. It is a significant burden on the programmer to identify the remote nodes in large systems, and we will return to this in Sections 4.2 and 5.1.2.

2.4. Scalability and Reliability in Erlang Systems

Erlang was designed to solve a particular set of problems, namely those in building telecommunications' infrastructure, where systems need to be scalable to accommodate hundreds of thousands of calls concurrently, in soft real-time. These systems need to be highly-available and reliable: i.e. to be robust in the case of failure, which can come from software or hardware faults. Given the inevitability of the latter, Erlang adopts the "let it fail" philosophy for error handling. That is, encourage programmers

[2]http://actor-framework.org
[3]http://pykka.readthedocs.org/en/latest/

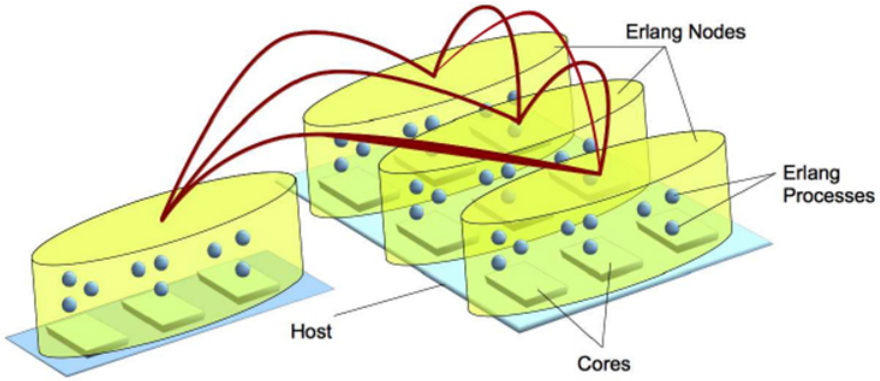

Fig. 1. Conceptual view of Erlang's concurrency, multicore support and distribution.

to embrace the fact that a process may fail at any point, and have them rely on the supervision mechanism, discussed shortly, to handle the failures.

Figure 1 illustrates Erlang's support for concurrency, multicores and distribution. Each Erlang node is represented by a yellow shape, and each rectangle represents a *host* with an IP address. Each red arc represents a *connection* between Erlang nodes. Each node can run on multiple cores, and exploit the inherent concurrency provided. This is done automatically by the VM, with no user intervention needed. Typically each core has an associated *scheduler* that schedules processes; a new process will be spawned on the same core as the process that spawns it, but work can be moved to a different scheduler through a work-stealing allocation algorithm. Each scheduler allows a process that is ready to compute at most a fixed number of computation steps before switching to another. Erlang *built-in functions* or BIFs are implemented in C, and at the start of the project were run to completion once scheduled, causing performance and responsiveness problems if the BIF had a long execution time.

Scaling in Erlang is provided in two different ways. It is possible to scale *within a single node* by means of the multicore virtual machine exploiting the concurrency provided by the multiple cores or NUMA nodes. It is also possible to scale *across multiple hosts* using multiple distributed Erlang nodes.

Reliability in Erlang is multi-faceted. As in all actor languages each process has private state, preventing a failed or failing process from corrupting the state of other processes. Messages enable stateful interaction, and contain a deep copy of the value to be shared, with no references (e.g. pointers) to the senders' internal state. Moreover Erlang avoids type errors by enforcing strong typing, albeit dynamically [Armstrong 2010]. Connected nodes check liveness with heartbeats, and can be monitored from outside Erlang, e.g. by an operating system process.

However, the most important way to achieve reliability is *supervision*, which allows a process to monitor the status of a child process and react to any failure, for example by spawning a substitute process to replace a failed process. Supervised processes can in turn supervise other processes, leading to a *supervision tree*. The supervising and supervised processes may be in different nodes, and on different hosts, and hence the supervision tree may span multiple hosts or nodes.

To provide reliable distributed service registration, a global namespace is maintained on every node, which maps *process names* to pids. It is this that we mean when we talk about a 'reliable' system: it is one in which a named process in a distributed

system can be restarted without requiring the client processes also to be restarted (because the name can still be used for communication).

To see global registration in action, consider a pong server process

```
global:register_name(pong_server, Remote_Pong_PID).
```

Clients of the server can send messages to the registered name, e.g.

```
global:whereis_name(pong_server) ! finish.
```

If the server fails the supervisor can spawn a replacement server process with a new pid and register it with the same name (`pong_server`). Thereafter client messages to the `pong_server` will be delivered to the new server process. We return to discuss the scalability limitations of maintaining a global namespace in Section 5.

2.5. ETS: Erlang Term Storage

Erlang is a pragmatic language and the actor model it supports is not pure. Erlang processes, besides communicating via asynchronous message passing, can also share data in public memory areas called *ETS tables*.

The Erlang Term Storage (ETS) mechanism is a central component of Erlang's implementation. It is used internally by many libraries and underlies the in-memory databases. ETS tables are key-value stores: they store Erlang tuples where one of the positions in the tuple serves as the lookup key. An ETS table has a *type* that may be either `set`, `bag` or `duplicate_bag`, implemented as a hash table, or `ordered_set` which is currently implemented as an AVL tree. The main operations that ETS supports are table creation, insertion of individual entries and atomic bulk insertion of multiple entries in a table, deletion and lookup of an entry based on some key, and destructive update. The operations are implemented as C built-in functions in the Erlang VM.

The code snippet below creates a `set` ETS table keyed by the first element of the entry; atomically inserts two elements with keys `some_key` and `42`; updates the value associated with the table entry with key `42`; and then looks up this entry.

```
Table = ets:new(my_table, [set, public, {keypos, 1}]),
ets:insert(Table, [{some_key, an_atom_value}, {42, {a,tuple,value}}]),
ets:update_element(Table, 42, [{2, {another,tuple,value}}]),
[{Key, Value}] = ets:lookup(Table, 42).
```

ETS tables are heavily used in many Erlang applications. This is partly due to the convenience of sharing data for some programming tasks, but also partly due to their fast implementation. As a shared resource, however, ETS tables induce contention and become a scalability bottleneck, as we shall see in Section 4.1.

3. BENCHMARKS FOR SCALABILITY AND RELIABILITY

The two benchmarks that we use throughout this article are Orbit, that measures scalability without looking at reliability, and Ant Colony Optimisation (ACO) that allows us to measure the impact on scalability of adding global namespaces to ensure reliability. The source code for the benchmarks, together with more detailed documentation, is available at https://github.com/release-project/benchmarks/. The RELEASE project team also worked to improve the reliability and scalability of other Erlang programs including a substantial (approximately 150K lines of Erlang code) Sim-Diasca simulator [Boudeville 2012] and an Instant Messenger that is more typical of Erlang applications [Chechina et al. 2016] but we do not cover these systematically here.

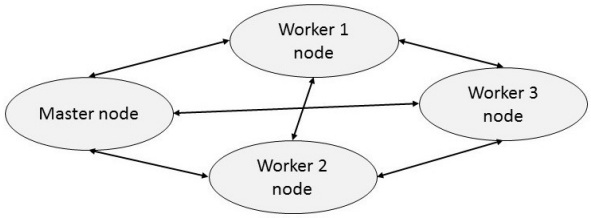

Fig. 2. Distributed Erlang Orbit (D-Orbit) architecture: workers are mapped to nodes.

3.1. Orbit

Orbit is a computation in symbolic algebra, which generalises a transitive closure computation [Lubeck and Neunhoffer 2001]. To compute the *orbit* for a given space $[0..X]$, a list of generators $g_1, g_2, ..., g_n$ are applied on an initial vertex $x_0 \in [0..X]$. This creates new values $(x_1...x_n) \in [0..X]$, where $x_i = g_i(x_0)$. The generator functions are applied on the new values until no new value is generated.

Orbit is a suitable benchmark because it has a number of aspects that characterise a class of real applications. The core data structure it maintains is a set and, in distributed environments is implemented as a *distributed hash table* (DHT), similar to the DHTs used in replicated form in NoSQL database management systems. Also, in distributed mode, it uses standard peer-to-peer (P2P) techniques like a credit-based termination algorithm [Matocha and Camp 1998]. By choosing the orbit size, the benchmark can be parameterised to specify smaller or larger computations that are suitable to run on a single machine (Section 4.1) or on many nodes (Section 8.1). Moreover it is only a few hundred lines of code.

As shown in Fig. 2, the computation is initiated by a master which creates a number of workers. In the single node scenario of the benchmark, workers correspond to processes but these workers can also spawn other processes to apply the generator functions on a subset of their input values, thus creating *intra-worker parallelism*. In the distributed version of the benchmark, processes are spawned by the master node to worker nodes, each maintaining a DHT fragment. A newly spawned process gets a share of the parent's credit, and returns this on completion. The computation is finished when the master node collects all credit, i.e. all workers have completed.

3.2. Ant Colony Optimisation (ACO)

Ant Colony Optimisation [Dorigo and Stützle 2004] is a meta-heuristic which has been applied to a large number of combinatorial optimisation problems. For the purpose of this article, we have applied it to an NP-hard scheduling problem known as the Single Machine Total Weighted Tardiness Problem (SMTWTP) [McNaughton 1959], where a number of jobs of given lengths have to be arranged in a single linear schedule. The goal is to minimise the *cost* of the schedule, as determined by certain constraints.

The ACO method is attractive from the point of view of distributed computing because it can benefit from having multiple cooperating colonies, each running on a separate compute node and consisting of multiple "ants". Ants are simple computational agents which concurrently compute possible solutions to the input problem guided by shared information about good paths through the search space; there is also a certain amount of stochastic variation which allows the ants to explore new directions. Having multiple colonies increases the number of ants, thus increasing the probability of finding a good solution.

We implement four distributed coordination patterns for the same multi-colony ACO computation as follows. In each implementation, the individual colonies perform some

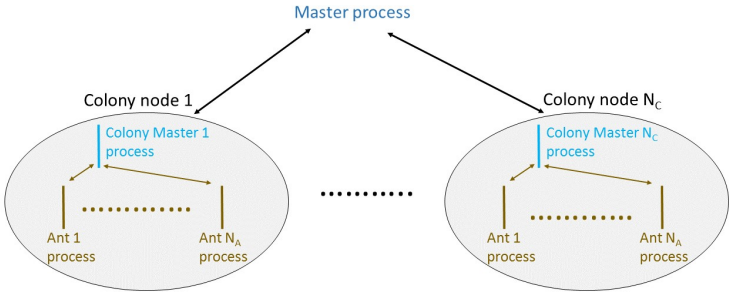

Fig. 3. Distributed Erlang Two-level Ant Colony Optimisation (TL-ACO) architecture.

number of *local iterations* (i.e. generations of ants) and then report their best solutions; the globally-best solution is then selected and is reported to the colonies, which use it to update their pheromone matrices. This process is repeated for some number of *global iterations*.

Two-level ACO (TL-ACO) has a single master node that collects the colonies' best solutions and distributes the overall best solution back to the colonies. Figure 3 depicts the process and node placements of the *TL-ACO* in a cluster with N_C nodes. The master process spawns N_C colony processes on available nodes. In the next step, each colony process spawns N_A ant processes on the local node. Each ant iterates I_A times, returning its result to the colony master. Each colony iterates I_M times, reporting their best solution to, and receiving the globally-best solution from, the master process. We validated the implementation by applying TL-ACO to a number of standard SMTWTP instances [Crauwels et al. 1998; Beasley 1990; Geiger 2010], obtaining good results in all cases, and confirmed that the number of perfect solutions increases as we increase the number of colonies.

Multi-level ACO (ML-ACO). In TL-ACO the master node receives messages from *all* of the colonies, and thus could become a bottleneck. ML-ACO addresses this by having a tree of submasters (Fig. 4), with each node in the bottom level collecting results from

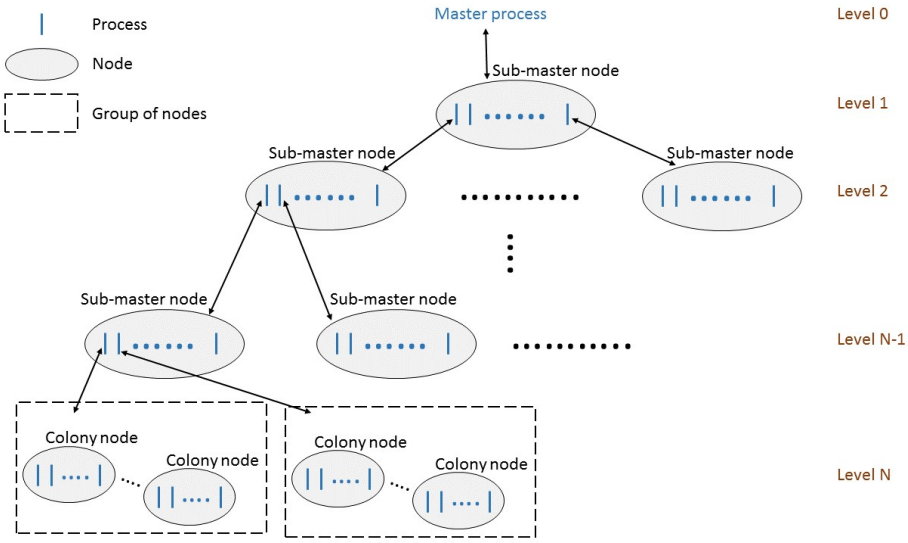

Fig. 4. Distributed Erlang Multi-level Ant Colony Optimisation (ML-ACO) architecture.

a small number of colonies. These are then fed up through the tree, with nodes at higher levels selecting the best solutions from their children.

Globally Reliable ACO (GR-ACO). In ML-ACO if a single colony fails to report back the system will wait indefinitely. GR-ACO adds fault tolerance, supervising colonies so that a faulty colony can be detected and restarted, allowing the system to continue execution.

Scalable Reliable ACO (SR-ACO) also adds fault-tolerance, but using supervision within our new s_groups from Section 5.1.1, and the architecture of SR-ACO is discussed in detail there.

4. ERLANG SCALABILITY LIMITS

This section investigates the scalability of Erlang at VM, language, and persistent storage levels. An aspect we choose not to explore is the security of large scale systems where, for example, one might imagine providing enhanced security for systems with multiple clusters or cloud instances connected by a Wide Area Network. We assume that existing security mechanisms are used, e.g. a Virtual Private Network.

4.1. Scaling Erlang on a Single Host

To investigate Erlang scalability we built BenchErl, an extensible open source benchmark suite with a web interface.[4] BenchErl shows how an application's performance changes when resources, like cores or schedulers, are added; or when options that control these resources change:

— the *number of nodes*, i.e. the number of Erlang VMs used, typically on multiple hosts;
— the *number of cores* per node;
— the *number of schedulers*, i.e. the OS threads that execute Erlang processes in parallel, and their *binding* to the topology of the cores of the underlying computer node;
— the *Erlang/OTP release and flavor*; and
— the *command-line arguments* used to start the Erlang nodes.

Using BenchErl, we investigated the scalability of an initial set of twelve benchmarks and two substantial Erlang applications using a single Erlang node (VM) on machines with up to 64 cores, including the Bulldozer machine specified in Appendix A. This set of experiments, reported by Aronis et al. [2012], confirmed that some programs scaled well in the most recent Erlang/OTP release of the time (R15B01) but also revealed VM and language level scalability bottlenecks.

Figure 5 shows runtime and speedup curves for the Orbit benchmark where a single master and 128 workers run on a single Erlang node in configurations with and without intra-worker parallelism. In both configurations the program scales. Runtime continuously decreases as we add more schedulers to exploit more cores. The speedup of the benchmark without intra-worker parallelism, i.e. without spawning additional processes for the computation that the workers perform (green curve), is almost linear up to 32 cores but increases less rapidly from that point on. This is due to the asymmetric characteristics of the machine's micro-architecture, which consists of modules that couple two conventional x86 out-of-order cores sharing the early pipeline stages, the floating point unit, and the L2 cache with the rest of the module [AMD 2015]. For the configuration with intra-worker parallelism (red curve) we see a similar but more clearly visible pattern. There is practically no performance improvement beyond 32 schedulers: spawing additional processes per worker for the computation is not benefi-

[4]Information about BenchErl is available at http://release.softlab.ntua.gr/bencherl/.

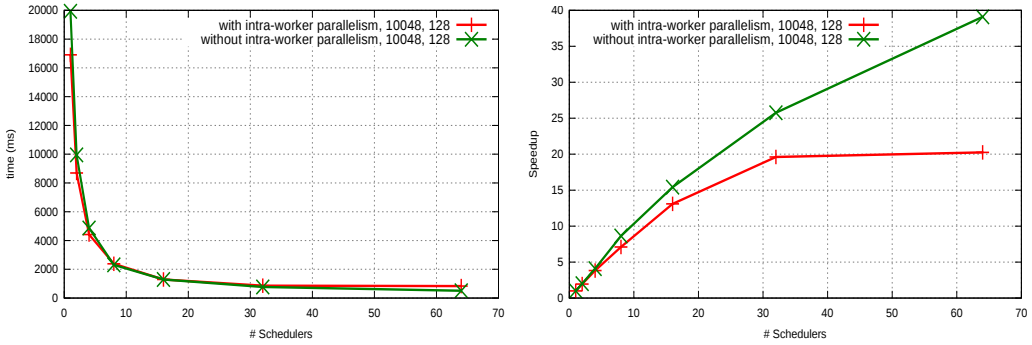

Fig. 5. Runtime and speedup of two configurations of the Orbit benchmark using Erlang/OTP R15B01.

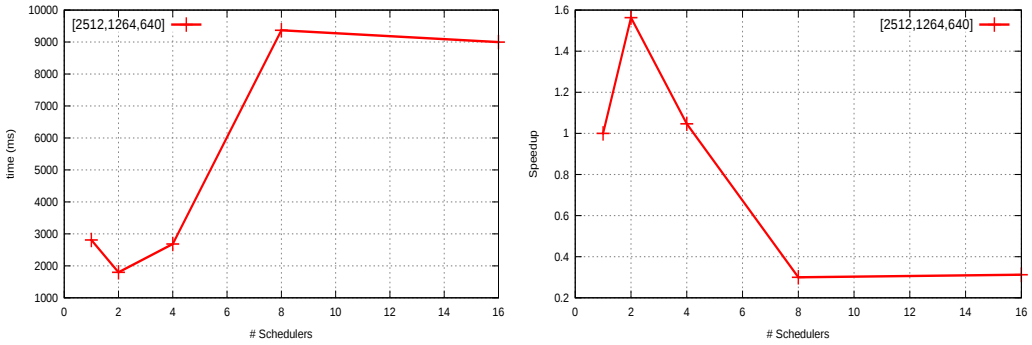

Fig. 6. Runtime and speedup of the ets_test BenchErl benchmark using Erlang/OTP R15B01.

cial for this set of benchmark parameters, where the number of workers (128) is higher than the number of cores.

Some other benchmarks, however, did not scale well or experienced significant slowdowns when run in many VM schedulers (threads). For example the ets_test benchmark has multiple processes accessing a shared ETS table. Figure 6 shows runtime and speedup curves for ets_test on a 16-core (eight cores with hyperthreading) Intel Xeon-based machine. It shows that runtime increases beyond two schedulers, and that the program exhibits a slowdown instead of a speedup.

For many benchmarks there are obvious reasons for poor scaling like limited parallelism in the application, or contention for shared resources. The reasons for poor scaling are less obvious for other benchmarks, and it is exactly these we have chosen to study in detail in subsequent work [Aronis et al. 2012; Klaftenegger et al. 2013].

A simple example is the parallel BenchErl benchmark, that spawns some n processes, each of which creates a list of m timestamps and, after it checks that each timestamp in the list is strictly greater than the previous one, sends the result to its parent. Figure 7 shows that up to eight cores each additional core leads to a slowdown, thereafter a small speedup is obtained up to 32 cores, and then again a slowdown. A small aspect of the benchmark, easily overlooked, explains the poor scalability. The benchmark creates timestamps using the `erlang:now/0` built-in function, whose implementation acquires a *global lock* in order to return a *unique timestamp*. That is, two calls to `erlang:now/0`, even from different processes are guaranteed to produce monotonically increasing values. This lock is precisely the bottleneck in the VM that limits

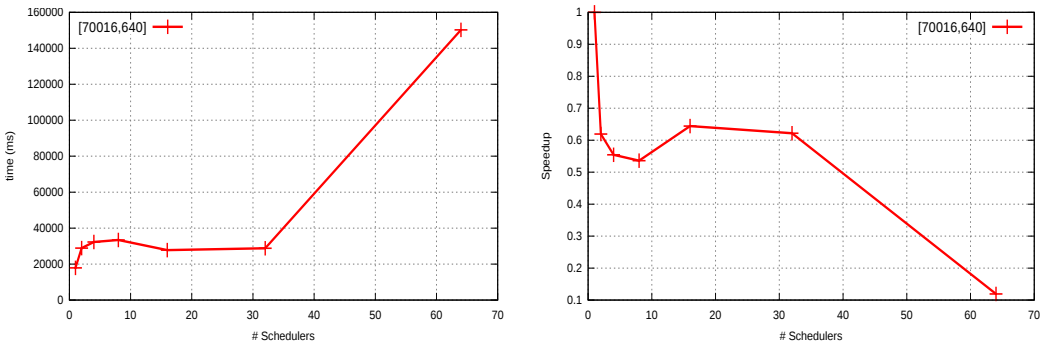

Fig. 7. Runtime and speedup of the BenchErl benchmark called parallel using Erlang/OTP R15B01.

the scalability of this benchmark. We describe our work to address VM timing issues in Section 6.3.

Discussion. Our investigations identified contention for shared ETS tables, and for commonly-used shared resources like timers, as the key VM-level scalability issues. Section 6 outlines how we addressed these issues in recent Erlang/OTP releases.

4.2. Distributed Erlang Scalability

Network Connectivity Costs. When any normal distributed Erlang nodes communicate, they share their connection sets and this typically leads to a fully connected graph of nodes. So a system with n nodes will maintain $O(n^2)$ connections, and these are relatively expensive TCP connections with continual maintenance traffic. This design aids transparent distribution as there is no need to discover nodes, and the design works well for small numbers of nodes. However at emergent server architecture scales, i.e. hundreds of nodes, this design becomes very expensive and system architects must switch from the default Erlang model, e.g. they need to start using hidden nodes that do not share connection sets.

We have investigated the scalability limits imposed by network connectivity costs using several Orbit calculations on two large clusters: Kalkyl and Athos as specified in Appendix A. The Kalkyl results are discussed by Chechina et al. [2016], and Fig. 28 in Section 8.1 shows representative results for distributed Erlang computing orbits with 2M and 5M elements on Athos. In all cases performance degrades beyond some scale (40 nodes for the 5M orbit, and 140 nodes for the 2M orbit). Figure 31 illustrates the additional network traffic induced by the fully connected network. It allows a comparison between the number of packets sent in a fully connected network (ML-ACO) with those sent in a network partitioned using our new s_groups (SR-ACO).

Global Information Costs. Maintaining global information is known to limit the scalability of distributed systems, and crucially the process namespaces used for reliability are global. To investigate the scalability limits imposed on distributed Erlang by such global information we have designed and implemented *DE-Bench*, an open source, parameterisable and scalable peer-to-peer benchmarking framework [Ghaffari 2014b; 2014a]. DE-Bench measures the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes, and the design is influenced by the Basho Bench benchmarking tool for Riak [Basho Technologies 2014]. Each DE-Bench instance acts as a peer, providing scalability and reliability by eliminating central coordination and any single point of failure.

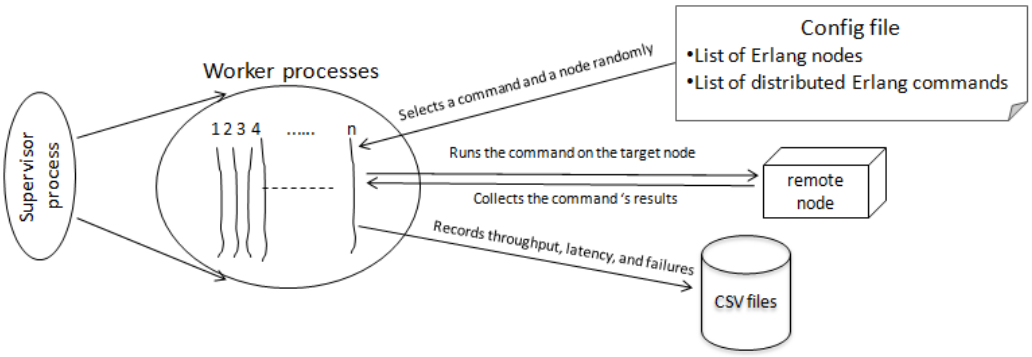

Fig. 8. DE-Bench's internal workflow.

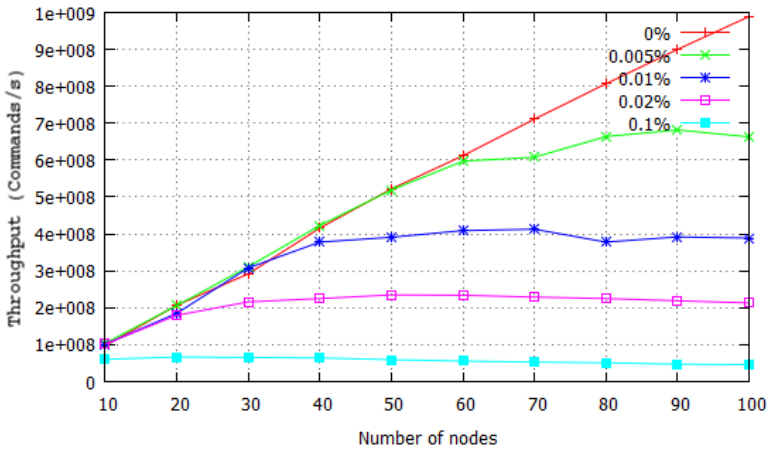

Fig. 9. Scalability vs. percentage of global commands in Distributed Erlang.

To evaluate the scalability of distributed Erlang, we measure how adding more hosts increases the throughput, i.e. the total number of successfully executed distributed Erlang commands per experiment. Figure 8 shows the parameterisable internal workflow of DE-Bench. There are three classes of commands in DE-Bench:(i) *Point-to-Point (P2P) commands*, where a function with tunable argument size and computation time is run on a remote node, include `spawn`, `rpc`, and synchronous calls to server processes, i.e. `gen_server` or `gen_fsm`. (ii) *Global commands*, which entail synchronisation across all connected nodes, such as `global:register_name` and `global:unregister_name`. (iii) *Local commands*, which are executed independently by a single node, e.g. `whereis_name`, a look up in the local name table.

The benchmarking is conducted on 10 to 100 host configurations of the Kalkyl cluster (in steps of 10) and measures the throughput of successful commands per second over 5 minutes. There is one Erlang VM on each host and one DE-Bench instance on each VM. The full paper [Ghaffari 2014b] investigates the impact of data size, and computation time in P2P calls both independently and in combination, and the scaling properties of the common Erlang/OTP generic server processes `gen_server` and `gen_fsm`.

Here we focus on the impact of different proportions of global commands, mixing global with P2P and local commands. Figure 9 shows that even a low proportion of

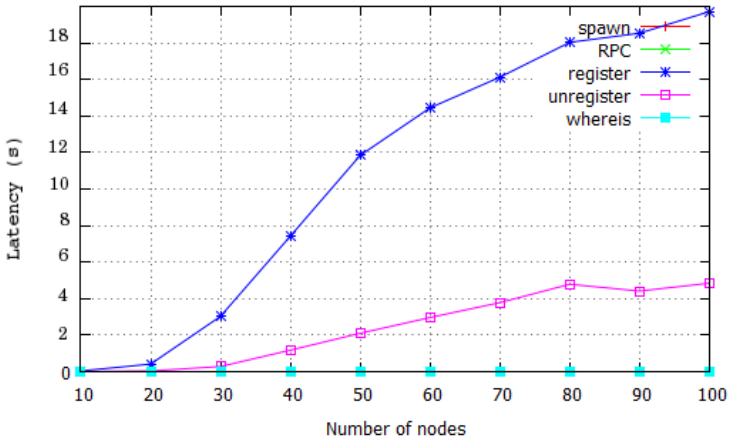

Fig. 10. Latency of commands as the number of Erlang nodes increases.

global commands limits the scalability of distributed Erlang, e.g. just 0.01% global commands limits scalability to around 40 nodes. Figure 10 reports the latency of all commands and shows that, while the latencies for P2P and local commands are stable at scale, the latency of the global commands increases dramatically with scale. Both results illustrate that the impact of global operations on throughput and latency in a distributed Erlang system is severe.

Explicit Placement. While network connectivity and global information impact *performance* at scale, our investigations also identified explicit process placement as a *programming* issue at scale. Recall from Section 2.3 that distributed Erlang requires the programmer to identify an explicit Erlang node (VM) when spawning a process. Identifying an appropriate node becomes a significant burden for large and dynamic systems. The problem is exacerbated in large distributed systems where (1) the hosts may not be identical, having different hardware capabilities or different software installed; and (2) communication times may be non-uniform: it may be fast to send a message between VMs on the same host, and slow if the VMs are on different hosts in a large distributed system.

These factors make it difficult to deploy applications, especially in a scalable and *portable* manner. Moreover while the programmer may be able to use platform-specific knowledge to decide where to spawn processes to enable an application to run efficiently, if the application is then deployed on a different platform, or if the platform changes as hosts fail or are added, this becomes outdated.

Discussion. Our investigations confirm three language-level scalability limitations of Erlang from developer folklore.(1) Maintaining a *fully connected network* of Erlang nodes limits scalability, for example Orbit is typically limited to just 40 nodes. (2) *Global operations*, and crucially the global operations required for reliability, i.e. to maintain a *global namespace*, seriously limit the scalability of distributed Erlang systems. (3) Explicit *process placement* makes it hard to built performance portable applications for large architectures. These issues cause designers of reliable large scale systems in Erlang to depart from the standard Erlang model, e.g. using techniques like hidden nodes and storing pids in data structures. In Section 5 we develop language technologies to address these issues.

4.3. Persistent Storage

Any large scale system needs reliable scalable persistent storage, and we have studied the scalability limits of Erlang persistent storage alternatives [Ghaffari et al. 2013]. We envisage a typical large server having around 10^5 cores on around 100 hosts. We have reviewed the requirements for scalable and available persistent storage and evaluated four popular Erlang DBMS against these requirements. For a target scale of around 100 hosts, Mnesia and CouchDB are, unsurprisingly, not suitable. However, Dynamo-style NoSQL DBMS like Cassandra and Riak have the potential to be.

We have investigated the current scalability limits of the Riak NoSQL DBMS using the Basho Bench benchmarking framework on a cluster with up to 100 nodes and independent disks. We found that that the scalability limit of Riak version 1.1.1 is 60 nodes on the Kalkyl cluster. The study placed into the public scientific domain what was previously well-evidenced, but anecdotal, developer experience.

We have also shown that resources like memory, disk, and network do not limit the scalability of Riak. By instrumenting the `global` and `gen_server` OTP libraries we identified a specific Riak remote procedure call that fails to scale. We outline how later releases of Riak are refactored to eliminate the scalability bottlenecks.

Discussion. We conclude that Dynamo-like NoSQL DBMSs have the potential to deliver reliable persistent storage for Erlang at our target scale of approximately 100 hosts. Specifically an Erlang Cassandra interface is available and Riak 1.1.1 already provides scalable and available persistent storage on 60 nodes. Moreover the scalability of Riak is much improved in subsequent versions.

5. IMPROVING LANGUAGE SCALABILITY

This section outlines the Scalable Distributed (SD) Erlang libraries [Chechina et al. 2016] we have designed and implemented to address the distributed Erlang scalability issues identified in Section 4.2. SD Erlang introduces two concepts to improve scalability. *S_groups* partition the set of nodes in an Erlang system to reduce network connectivity and partition global data (Section 5.1.1). *Semi-explicit placement* alleviates the issues of explicit process placement in large heterogeneous networks (Section 5.1.2). The two features are independent and can be used separately or in combination. We overview SD Erlang in Section 5.1, and outline s_group semantics and validation in Sections 5.2 and 5.3 respectively.

5.1. SD Erlang Design

5.1.1. S_groups. reduce both the number of connections a node maintains, and the size of name spaces, i.e. they minimise global information. Specifically names are registered on, and synchronised between, only the nodes within the s_group. An s_group has the following parameters: a name, a list of nodes, and a list of registered names. A node can belong to many s_groups or to none. If a node belongs to no s_group it behaves as a usual distributed Erlang node.

The `s_group` library defines the functions shown in Table I. Some of these functions manipulate s_groups and provide information about them, such as creating s_groups and providing a list of nodes from a given s_group. The remaining functions manipulate names registered in s_groups and provide information about these names. For example, to register a process, `Pid`, with name `Name` in s_group `SGroupName` we use the following function. The name will only be registered if the process is being executed on a node that belongs to the given s_group, and neither `Name` nor `Pid` are already registered in that group.

```
s_group:register_name(SGroupName, Name, Pid) -> yes | no
```

Table I. Summary of `s_group` functions.

Function	Description
`new_s_group(SGroupName, Nodes)`	Creates a new s_group consisting of some nodes.
`delete_s_group(SGroupName)`	Deletes an s_group.
`add_nodes(SGroupName, Nodes)`	Adds a list of nodes to an s_group.
`remove_nodes(SGroupName, Nodes)`	Removes a list of nodes from an s_group.
`s_groups()`	Returns a list of all s_groups known to the node.
`own_s_groups()`	Returns a list of s_group tuples the node belongs to.
`own_nodes()`	Returns a list of nodes from all s_groups the node belongs to.
`own_nodes(SGroupName)`	Returns a list of nodes from the given s_group.
`info()`	Returns s_group state information.
`register_name(SGroupName, Name, Pid)`	Registers a name in the given s_group.
`re_register_name(SGroupName, Name, Pid)`	Re-registers a name (changes a registration) in a given s_group.
`unregister_name(SGroupName, Name)`	Unregisters a name in the given s_group.
`registered_names({node,Node})`	Returns a list of all registered names on the given node.
`registered_names({s_group,SGroupName})`	Returns a list of all registered names in the given s_group.
`whereis_name(SGroupName, Name)` `whereis_name(Node, SGroupName, Name)`	Return the pid of a name registered in the given s_group.
`send(SGroupName, Name, Msg)` `send(Node, SGroupName, Name, Msg)`	Send a message to a name registered in the given s_group.

To illustrate the impact of s_groups on scalability we repeat the global operations experiment from Section 4.2 (Fig. 9). In the SD Erlang experiment we partition the set of nodes into s_groups each containing ten nodes, and hence the names are replicated and synchronised on just ten nodes, and not on all nodes as in distributed Erlang. The results in Fig. 11 show that with 0.01% of global operations throughput of distributed Erlang stops growing at 40 nodes while throughput of SD Erlang continues to grow linearly.

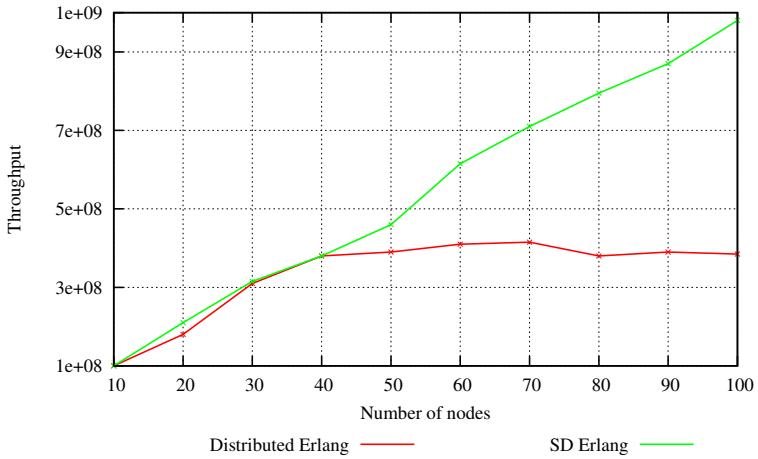

Fig. 11. Global operations in Distributed Erlang vs. SD Erlang.

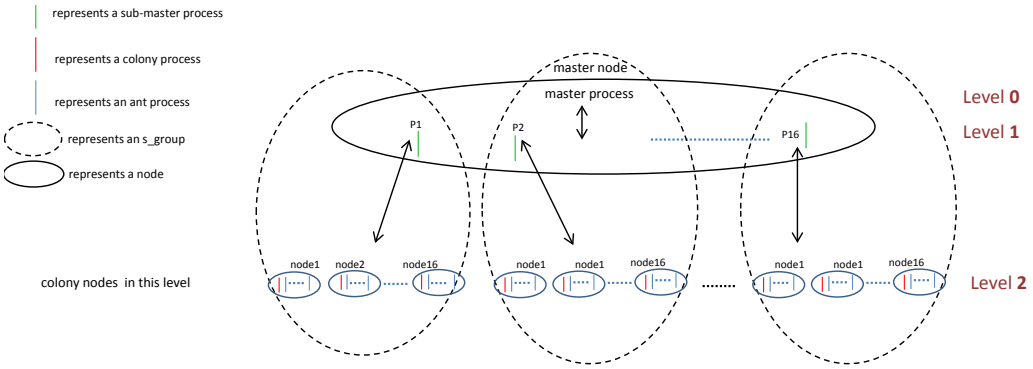

Fig. 12. SD Erlang ACO (SR-ACO) architecture.

The connection topology of s_groups is extremely flexible: they may be organised into a hierarchy of arbitrary depth or branching, e.g. there could be multiple levels in the tree of s_groups; see Fig. 12. Moreover it is not necessary to create an hierarchy of s_groups, for example, we have constructed an Orbit implementation using a ring of s_groups.

Given such a flexible way of organising distributed systems, key questions in the design of an SD Erlang system are the following. *How should s_groups be structured?* Depending on the reason the nodes are grouped – reducing the number of connections, or reducing the namespace, or both – s_groups can be freely structured as a tree, ring, or some other topology. *How large should the s_groups be?* Smaller s_groups mean more inter-group communication, but the synchronisation of the s_group state between the s_group nodes constrains the maximum size of s_groups. We have not found this constraint to be a serious restriction. For example many s_groups are either relatively small, e.g. 10-node, internal or terminal elements in some topology, e.g. the leaves and nodes of a tree. *How do nodes from different s_groups communicate?* While any two nodes can communicate in an SD Erlang system, to minimise the number of connections communication between nodes from different s_groups is typically routed via gateway nodes that belong to both s_groups. *How do we avoid single points of failure?* For reliability, and to minimise communication load, multiple gateway nodes and processes may be required.

Information to make these design choices is provided by the tools in Section 7 and by benchmarking. A further challenge is how to systematically refactor a distributed Erlang application into SD Erlang, and this is outlined in Section 7.1. A detailed discussion of distributed system design and refactoring in SD Erlang provided in a recent article [Chechina et al. 2017].

We illustrate typical SD Erlang system designs by showing refactorings of the Orbit and ACO benchmarks from Section 3. In both distributed and SD Erlang the computation starts on the *Master* node and the actual computation is done on the *Worker* nodes. In the distributed Erlang version all nodes are interconnected, and messages are transferred directly from the sending node to the receiving node (Fig. 2). In contrast, in the SD Erlang version nodes are grouped into s_groups, and messages are transferred between different s_groups via *Sub-master* nodes (Fig. 13).

A fragment of code that creates an s_group on a *Sub-master* node is as follows:

```
create_s_group(Master, GroupName, Nodes0) ->
  case s_group:new_s_group(GroupName, Nodes0) of
    {ok, GroupName, Nodes} -> Master ! {GroupName, Nodes};
```

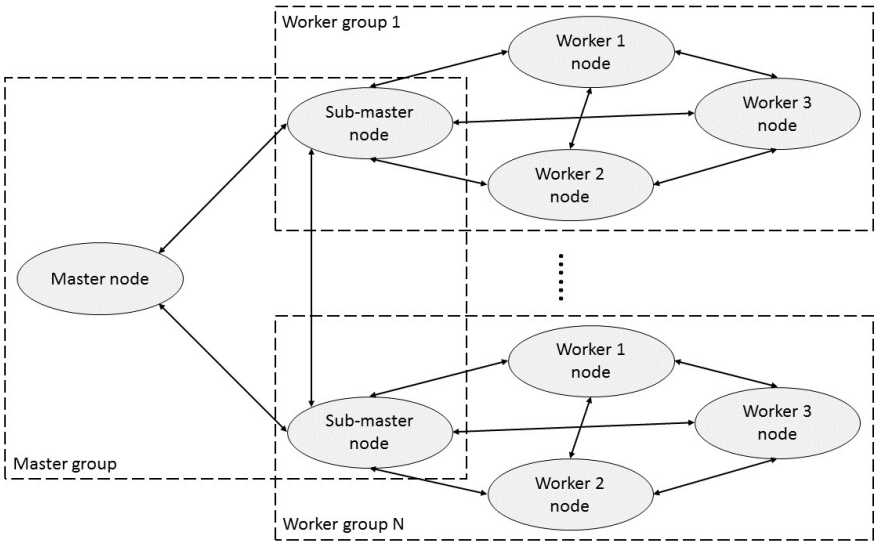

Fig. 13. SD Erlang (SD-Orbit) architecture.

```
  _ -> io:format("exception: ... some message ...")
end.
```

Similarly, we introduce s_groups in the GR-ACO benchmark from Section 3.2 to create Scalable Reliable ACO (SR-ACO); see Fig. 12. Here, apart from reducing the number of connections, s_groups also reduce the global namespace information. That is, instead of registering the name of a pid globally, i.e. with all nodes, the names is registered only on all nodes in the s_group with `s_group:register_name/3`.

A comparative performance evaluation of distributed Erlang and SD Erlang Orbit and ACO is presented in Section 8.

5.1.2. Semi-Explicit Placement. Recall from Section 2.3 that distributed Erlang spawns a process onto an explicitly named Erlang node, e.g.

```
spawn(some_node, fun some_module:pong/0).
```

and also recall the portability and programming effort issues associated with such explicit placement in large scale systems discussed in Section 4.2.

To address these issues we have developed a semi-explicit placement library that enables the programmer to select nodes on which to spawn processes based on runtime information about the properties of the nodes. For example, if a process performs a lot of computation one would like to spawn it on a node with considerable computation power, or if two processes are likely to communicate frequently then it would be desirable to spawn them on the same node, or nodes with a fast interconnect.

We have implemented two Erlang libraries to support semi-explicit placement [MacKenzie et al. 2015]. The first deals with *node attributes*, and describes properties of individual Erlang VMs and associated hosts, such as total and currently available RAM, installed software, hardware configuration, etc. The second deals with a notion of *communication distances* which models the communication times between nodes in a distributed system. Therefore, instead of specifying a node we can use the `attr:choose_node/1` function to define the target node, i.e.

```
spawn(attr:choose_node(Params), fun some_module:pong/0).
```

$$
\begin{aligned}
(grs, fgs, fhs, nds) \in \{state\} &\equiv \big\{\big(\{s\_group\}, \{free\_group\}, \{free\_hidden\_group\}, \{node\}\big)\big\} \\
gr \in grs \equiv \{s\_group\} &\equiv \big\{\big(s\_group\_name, \{node\_id\}, namespace\big)\big\} \\
fg \in fgs \equiv \{free\_group\} &\equiv \big\{\big(\{node\_id\}, namespace\big)\big\} \\
fh \in fhs \equiv \{free\_hidden\_group\} &\equiv \big\{\big(node\_id, namespace\big)\big\} \\
nd \in nds \equiv \{node\} &\equiv \big\{\big(node\_id, node\_type, connections, gr\_names\big)\big\} \\
gs \in \{gr\_names\} &\equiv \big\{NoGroup, \{s\_group\_name\}\big\} \\
ns \in \{namespace\} &\equiv \big\{\{(name, pid)\}\big\} \\
cs \in \{connections\} &\equiv \big\{\{node\_id\}\big\} \\
nt \in \{node\_type\} &\equiv \{Normal, Hidden\} \\
s \in \{NoGroup, s\_group\_name\}
\end{aligned}
$$

Fig. 14. SD Erlang state [Chechina et al. 2016].

MacKenzie et al. [2015] report an investigation into the communication latencies on a range of NUMA and cluster architectures, and demonstrate the effectiveness of the placement libraries using the ML-ACO benchmark on the Athos cluster.

5.2. S_group Semantics

For precise specification, and as a basis for validation, we provide a small-step operational semantics of the s_group operations [Chechina et al. 2016]. Figure 14 defines the state of an SD Erlang system and associated abstract syntax variables. The abstract syntax variables on the left are defined as members of sets, denoted $\{\}$, and these in turn may contain tuples, denoted $()$, or further sets. In particular nm is a process name, p a pid, ni a node_id, and nis a set of node_ids. The state of a system is modelled as a four tuple comprising a set of s_groups, a set of $free_group$s, a set of $free_hidden_group$s, and a set of $node$s. Each type of group is associated with nodes and has a namespace. An s_group additionally has a name, whereas a $free\_hidden\_group$ consists of only one node, i.e. a hidden node simultaneously acts as a node and as a group, because as a group it has a namespace but does not share it with any other node. Free normal and hidden groups have no names, and are uniquely defined by the nodes associated with them. Therefore, group names, gr_names, are either $NoGroup$ or a set of s_group_names. A $namespace$ is a set of $name$ and process id, pid, pairs and is replicated on all nodes of the associated group.

A $node$ has the following four parameters: $node\_id$ identifier, $node\_type$ that can be either hidden or normal, $connections$, and $group\_names$, i.e. names of groups the node belongs to. The node can belong to either a list of s_groups or one of the free groups. The type of the free group is defined by the node type. Connections are a set of $node_id$s.

Transitions in the semantics have the form $(state, command, ni) \longrightarrow (state', value)$ meaning that executing $command$ on node ni in $state$ returns $value$ and transitions to $state'$.

The semantics is presented in more detail by Chechina et al. [2016], but we illustrate it here with the `s_group:registered_names/1` function from Section 5.1.1. The function returns a list of names registered in s_group s if node ni belongs to the s_group, an empty list otherwise.

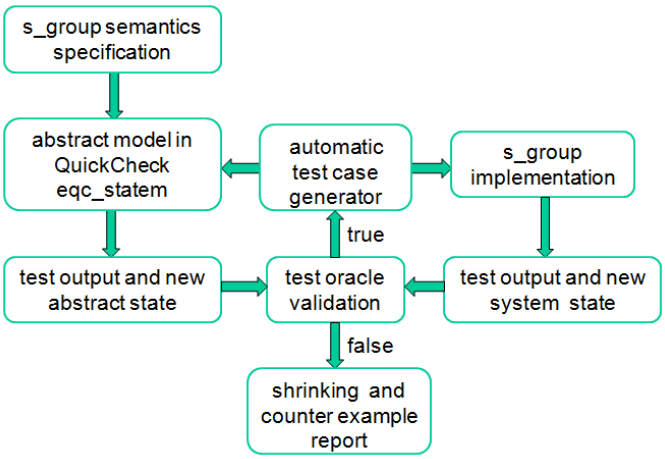

Fig. 15. Testing s_groups using QuickCheck.

$$
\begin{aligned}
&\big((grs, fgs, fhs, nds), \text{s\_group} : \text{registered\_names(s)}, ni\big) \\
&\quad \longrightarrow \big((grs, fgs, fhs, nds), nms\big) && \textit{if } \textit{IsSGroupNode}(ni, s, grs) \\
&\quad \longrightarrow \big((grs, fgs, fhs, nds), \{\}\big) && \textit{otherwise} \\
&\textit{where} \\
&\qquad \big\{(s, \{ni\} \oplus nis, ns)\big\} \oplus grs' \equiv grs \\
&\qquad nms \equiv \textit{OutputNms}(s, ns)
\end{aligned}
$$

Here $\oplus$ denotes disjoint set union; *IsSGroupNode* returns true if node ni is a member of some s_group s, false otherwise; and *OutputNms* returns a set of process names registered in the ns namespace of s_group s.

$$\textit{IsSGroupNode}(ni, s, grs) = \exists nis, ns, grs' \,.\, \big\{(s, \{ni\} \oplus nis, ns)\big\} \oplus grs' \equiv grs$$

$$\textit{OutputNms}(s, ns) = \big\{(s, nm) \mid (nm, p) \in ns\big\}$$

5.3. Semantics Validation

As the semantics is concrete it can readily be made *executable* in Erlang, with lists replacing sets throughout. Having an executable semantics allows users to engage with it, and to understand how the semantics behaves *vis à vis* the library, giving them an opportunity to assess the correctness of the library against the semantics.

Better still, we can automatically assess how the system behaves in comparison with the (executable) semantics by executing them in lockstep, guided by the constraints of which operations are possible at each point. We do that by building an abstract state machine model of the library. We can then generate random sequences (or traces) through the model, with appropriate library data generated too. This random generation is supported by the QuickCheck property-based testing system [Claessen and Hughes 2000; Arts et al. 2006].

The architecture of the testing framework is shown in Fig. 15. First an abstract state machine embedded as an "`eqc_statem`" module is derived from the executable semantic specification. The state machine defines the abstract state representation and the transition from one state to another when an operation is applied. Test case and data generators are then defined to control the test case generation; this includes the automatic generation of eligible s_group operations and the input data to those operations. Test oracles are encoded as the postcondition for s_group operations.

During testing, each test command is applied to both the abstract model and the s_group library. The application of the test command to the abstract model takes the abstract model from its current state to a new state as described by the transition functions; whereas the application of the test command to the library leads the system to a new actual state. The actual state information is collected from each node in the distributed system, then merged and normalised to the same format as the abstract state representation. For a test to be successful, after the execution of a test command, the test oracles specified for this command should be satisfied. Various test oracles can be defined for s_group operations; for instance one of the generic constraints that applies to all the s_group operations is that after each s_group operation, the normalised system state should be equivalent to the abstract state.

Thousands of tests were run, and three kinds of errors — which have subsequently been corrected — were found. Some errors in the library implementation were found, including one error due to the synchronisation between nodes, and the other related to the `remove_nodes` operation, which erroneously raised an exception. We also found a couple of trivial errors in the semantic specification itself, which had been missed by manual examination. Finally, we found some situations where there were inconsistencies between the semantics and the library implementation, despite their states being equivalent: an example of this was in the particular values returned by functions on certain errors. Overall, the automation of testing boosted our confidence in the correctness of the library implementation and the semantic specification. This work is reported in more detail by Li and Thompson [2014].

6. IMPROVING VM SCALABILITY

This section reports the primary VM and library improvements we have designed and implemented to address the scalability and reliability issues identified in Section 4.1.

6.1. Improvements to Erlang Term Storage

Because ETS tables are so heavily used in Erlang systems, they are a focus for scalability improvements. We start by describing their redesign, including some improvements that pre-date our RELEASE project work, i.e. those prior to Erlang/OTP R15B03. These historical improvements are very relevant for a scalability study and form the basis for our subsequent changes and improvements. At the point when Erlang/OTP got support for multiple cores (in release R11B), there was a single reader-writer lock for each ETS table. Optional fine grained locking of hash-based ETS tables (i.e. `set`, `bag` or `duplicate_bag` tables) was introduced in Erlang/OTP R13B02-1, adding 16 reader-writer locks for the hash buckets. Reader groups to minimise read synchronisation overheads were introduced in Erlang/OTP R14B. The key observation is that a single count of the multiple readers must be synchronised across many cache lines, potentially far away in a NUMA system. Maintaining reader counts in multiple (local) caches makes reads fast, although writes must now check every reader count. In Erlang/OTP R16B the number of bucket locks, and the default number of reader groups, were both upgraded from 16 to 64.

We illustrate the scaling properties of the ETS concurrency options using the ets_bench BenchErl benchmark on the Intel NUMA machine with 32 hyperthreaded

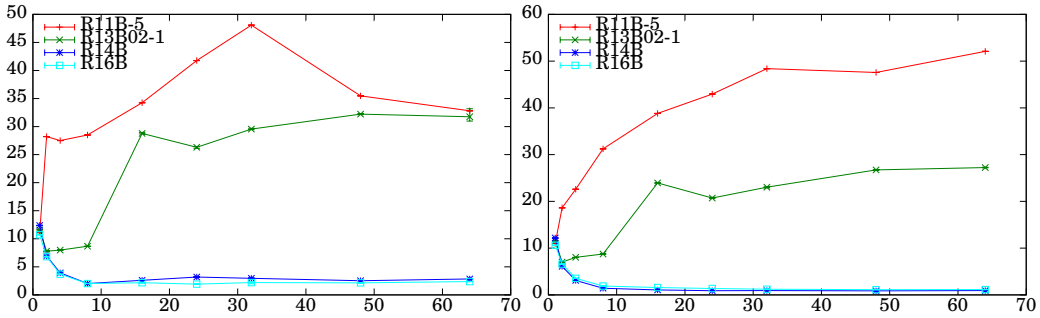

Fig. 16. Runtime of 17M ETS operations in Erlang/OTP releases: 10% updates (left) and 1% updates (right). The y-axis shows time in seconds (lower is better) and the x-axis is number of OS threads (VM schedulers).

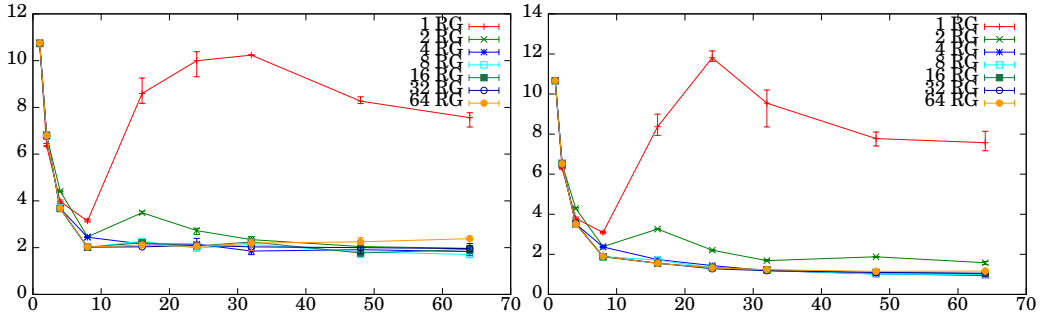

Fig. 17. Runtime of 17M ETS operations with varying numbers of reader groups: 10% updates (left) and 1% updates (right). The y-axis shows runtime in seconds (lower is better) and the x-axis is number of schedulers.

cores specified in Appendix A. The ets_bench benchmark inserts 1M items into the table, then records the time to perform 17M operations, where an operation is either a lookup, an insert, or a delete. The experiments are conducted on a hash-based (`set`) ETS table with different percentages of update operations, i.e. insertions or deletions.

Figure 16 shows the runtimes in seconds of 17M operations in different Erlang/OTP versions, varying the number of schedulers (x-axis), reflecting how the scalability of ETS tables has improved in more recent Erlang/OTP releases. Figure 17 shows the runtimes in seconds of 17M operations on an ETS table with different numbers of reader groups, again varying the number of schedulers. We see that one reader group is not sufficient with 10% updates, nor are two with 1% updates. Beyond that, different numbers of reader groups have little impact on the benchmark performance except that using 64 groups with 10% updates slightly degrades performance.

We have explored four other extensions or redesigns in the ETS implementation for better scalability. (1) Allowing more programmer control over the number of bucket locks in hash-based tables, so the programmer can reflect the number of schedulers and the expected access pattern. (2) Using contention-adapting trees to get better scalability for `ordered_set` ETS tables as described by Sagonas and Winblad [2014]. (3) Using queue delegation locking to improve scalability [Klaftenegger et al. 2014]. (4) Adopting schemes for completely eliminating the locks in the meta table. A more complete discussion of our work on ETS can be found in the papers by Sagonas and Winblad [2014] and Klaftenegger et al. [2014].

Here we outline only our work on contention-adapting (CA) trees. A CA tree monitors contention in different parts of a tree-shaped data structure, introducing routing nodes with locks in response to high contention, and removing them in response to

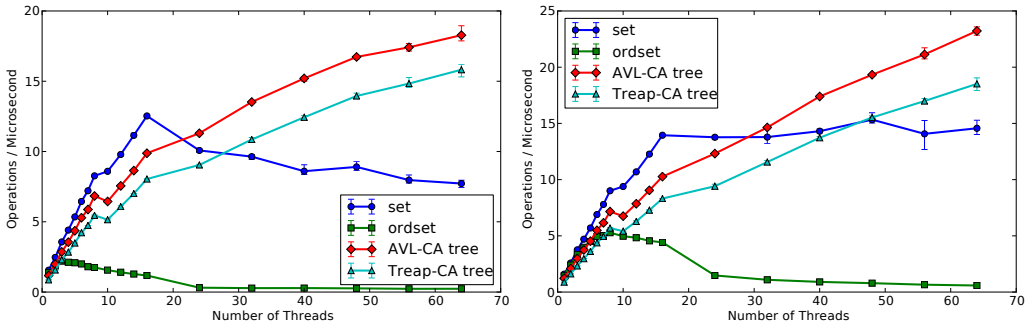

Fig. 18. Throughput of CA tree variants: 10% updates (left) and 1% updates (right).

low contention. For experimental purposes two variants of the CA tree have been implemented to represent `ordered_sets` in the virtual machine of Erlang/OTP 17.0. One extends the existing AVL trees in the Erlang VM, and the other uses a *Treap* data structure [Aragon and Seidel 1989]. Figure 18 compares the throughput of the CA tree variants with that of `ordered_set` and `set` as the number of schedulers increases. It is unsurprising that the CA trees scale so much better than an `ordered_set` which is protected by a single readers-writer lock. It is more surprising that they also scale better than `set`. This is due to hash tables using fine-grained locking at a fixed granularity, while CA trees can adapt the number of locks to the current contention level, and also to the parts of the key range where contention is occurring.

6.2. Improvements to Schedulers

In the Erlang VM, a scheduler is responsible for executing multiple processes concurrently, in a timely and fair fashion, making optimal use of hardware resources. The VM implements preemptive multitasking with soft real-time guarantees. Erlang processes are normally scheduled on a reduction count basis where one reduction is roughly equivalent to a function call. Each process is allowed to execute until it either blocks waiting for input, typically a message from some other process, or until it has executed its quota of reductions.

The Erlang VM is usually started with one scheduler per logical core (SMT-thread) available on the host machine, and schedulers are implemented as OS threads. When an Erlang process is spawned it is placed in the run queue of the scheduler of its parent, and it waits on that queue until the scheduler allocates it a slice of core time. Work stealing is used to balance load between cores, that is an idle scheduler may *migrate* a process from another run queue. Scheduler run queues are visualised in Section 7.3.2.

The default load management mechanism is load compaction that aims to keep as many scheduler threads as possible fully loaded with work, i.e. it attempts to ensure that scheduler threads do not run out of work. We have developed a new optional *scheduler utilisation balancing* mechanism that is available from Erlang/OTP 17.0. The new mechanism aims to balance scheduler utilisation between schedulers; that is, it will strive for equal scheduler utilisation on all schedulers.

The scheduler utilisation balancing mechanism has no performance impact on the system when not enabled. On the other hand, when enabled, it results in changed timing in the system; normally there is a small overhead due to measuring of utilisation and calculating balancing information, which depends on the underlying primitives provided by the operating system.

The new balancing mechanism results in a better distribution of processes to schedulers, reducing the probability of core contention. Together with other VM improve-

ments, such as interruptable BIFs and garbage collection, it results in lower latency and improved responsiveness, and hence reliability, for soft real-time applications.

6.3. Improvements to Time Management

Soon after the start of the RELEASE project, time management in the Erlang VM became a scalability bottleneck for many applications, as illustrated by the parallel benchmark in Section 4.1. The issue came to prominence as other, more severe, bottlenecks were eliminated. This subsection motivates and outlines the improvements to time management that we made; these were incorporated into Erlang/OTP 18.x as a new API for time and time warping. The old API is still supported at the time of writing, but its use is deprecated.

The original time API provides the `erlang:now/0` built-in that returns "Erlang system time" or time since Epoch with micro second resolution. This time is the basis for all time internally in the Erlang VM.

Many of the scalability problems of `erlang:now/0` stem from its specification, written at a time when the Erlang VM was not multi-threaded, i.e. SMT-enabled. The documentation promises that values returned by it are strictly increasing and many applications ended up relying on this. For example applications often employ `erlang:now/0` to generate unique integers.

Erlang system time should align with the operating system's view of time since Epoch or "OS system time". However, while OS system time can be freely changed both forwards and backwards, Erlang system time cannot, without invalidating the strictly increasing value guarantee. The Erlang VM therefore contains a mechanism that slowly adjusts Erlang system time towards OS system time if they do not align.

One problem with time adjustment is that the VM deliberately presents time with an inaccurate frequency; this is required to align Erlang system time with OS system time smoothly when these two have deviated, e.g. in the case of clock shifts when leap seconds are inserted or deleted. Another problem is that Erlang system time and OS system time can differ for very long periods of time. In the new API, we resolve this using a common OS technique [LWN.net 2006], i.e. 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. Erlang system time is thus just a dynamically varying offset from Erlang monotonic time.

Time Retrieval. Retrieval of Erlang system time was previously protected by a global mutex, which made the operation thread safe, but scaled poorly. 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 the machine's local clock and cannot be changed, while the system time is adjusted using the Network Time Protocol (NTP). That is, they will run with different frequencies. Linux is an exception with a monotonic clock that is NTP adjusted and runs with the same frequency as system time [Völker 2014]. To align the frequencies of Erlang monotonic time and Erlang system time, we adjust the frequency of the Erlang monotonic clock. This is done by comparing monotonic time and system time delivered by the OS, and calculating an adjustment. To achieve this scalably, one VM thread calculates the time adjustment to use at least once a minute. If the adjustment needs to be changed, new adjustment information is published and used to calculate Erlang monotonic time in the future.

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. To preserve monotonicity it is important that all threads that read the same OS monotonic time map this to exactly the same Erlang monotonic time. This

requires synchronisation on updates to the adjustment information using a readers-writer (RW) lock. This 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. To prevent bouncing the lock cache-line we use a bespoke reader optimised RW lock implementation where reader threads notify about their presence in counters on separate cache-lines. The concept is similar to the reader indicator algorithm described by Klaftenegger et al. [2017, Fig. 11] and alternatives include the ingress-egress counter used by Calciu et al. [2013] and the SNZI algorithm of Ellen et al. [2007].

Timer Wheel and BIF Timer. The timer wheel contains all timers set by Erlang processes. The original implementation was protected by a global mutex and scaled poorly. To increase concurrency, each scheduler thread has been assigned its own timer wheel that is used by processes executing on the scheduler.

The implementation of timers in Erlang/OTP uses a built in function (BIF), as most low-level operations do. Until Erlang/OTP 17.4, this BIF was also protected by a global mutex. Besides inserting timers into the timer wheel, the BIF timer implementation also maps timer references to a timer in the timer wheel. To improve concurrency, from Erlang/OTP 18 we provide scheduler-specific BIF timer servers as Erlang processes. These keep information about timers in private ETS tables and only insert one timer at the time into the timer wheel.

Benchmarks. We have measured several benchmarks on a 16-core Bulldozer machine with eight dual CPU AMD Opteron 4376 HEs.[5] We present three of them here.

The first micro benchmark compares the execution time of an Erlang `receive` with that of a `receive after` that specifies a timeout and provides a default value. The `receive after` sets a timer when the process blocks in the `receive`, and cancels it when a message arrives. In Erlang/OTP 17.4 the total execution time with standard timers is 62% longer than without timers. Using the improved implementation in Erlang/OTP 18.0, total execution time with the optimised timers is only 5% longer than without timers.

The second micro benchmark repeatedly checks the system time, calling the built-in `erlang:now/0` in Erlang/OTP 17.4, and calling both `erlang:monotonic_time/0` and `erlang:time_offset/0` and adding the results in Erlang/OTP 18.0. In this machine, where the VM uses 16 schedulers by default, the 18.0 release is more than 69 times faster than the 17.4 release.

The third benchmark is the parallel BenchErl benchmark from Section 4.1. Figure 19 shows the results of executing the original version of this benchmark, which uses `erlang:now/0` to create monotonically increasing unique values, using three Erlang/OTP releases: R15B01, 17.4, and 18.1. We also measure a version of the benchmark in Erlang/OTP 18.1 where the call to `erlang:now/0` has been substituted with a call to `erlang:monotonic_time/0`. The graph on its left shows that: (1) the performance of time management has remained roughly unchanged between Erlang/OTP releases prior to 18.0; (2) the improved time management in Erlang/OTP 18.x make time management less likely to be a scalability bottleneck even when using `erlang:now/0`, and (3) the new time API (using `erlang:monotonic_time/0` and friends) provides a scalable solution. The graph on the right side of Fig. 19 shows the speedup that the modified version of the parallel benchmark achieves in Erlang/OTP 18.1.

[5]See §2.5.4 of the RELEASE project Deliverable 2.4 (http://release-project.eu/documents/D2.4.pdf).

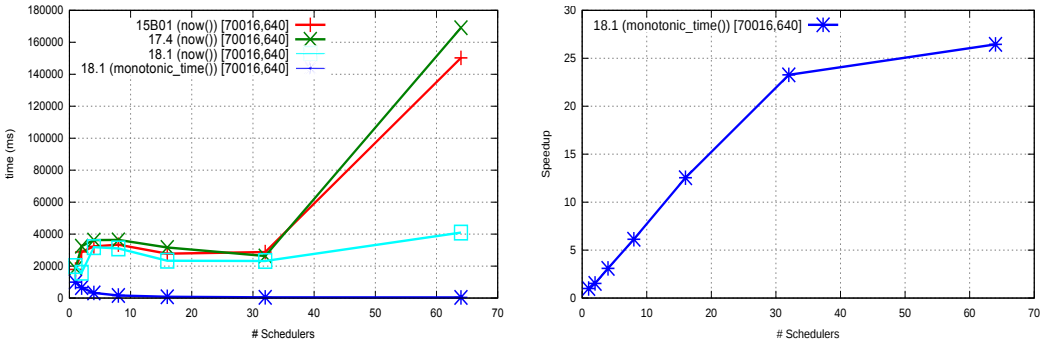

Fig. 19. BenchErl parallel benchmark using `erlang:monotonic_time/0` or `erlang:now/0` in different Erlang/OTP releases: runtimes (left) and speedup obtained using `erlang:monotonic_time/0` (right).

7. SCALABLE TOOLS

This section outlines five tools developed in the RELEASE project to support scalable Erlang systems. Some tools were developed from scratch, like Devo, SDMon and WombatOAM, while others extend existing tools, like Percept and Wrangler. These include tooling to transform programs to make them more scalable, to deploy them for scalability, to monitor and visualise them. Most of the tools are freely available under open source licences (Devo, Percept2, SD-Mon, Wrangler); while WombatOAM is a commercial product. The tools have been used for profiling and refactoring the ACO and Orbit benchmarks from Section 3.

The Erlang tool "ecosystem" consists of small stand-alone tools for tracing, profiling and debugging Erlang systems that can be used separately or together as appropriate for solving the problem at hand, rather than as a single, monolithic, super-tool. The tools presented here have been designed to be used as part of that ecosystem, and to complement already available functionality rather than to duplicate it. The Erlang runtime system has built-in support for tracing many types of events, and this infrastructure forms the basis of a number of tools for tracing and profiling. Typically the tools build on or specialise the services offered by the Erlang virtual machine, through a number of built-in functions. Most recently, and since the RELEASE project was planned, the Observer[6] application gives a comprehensive overview of many of these data on a node-by-node basis.

As actor frameworks and languages (see Section 2.2) have only recently become widely adopted commercially, their tool support remains relatively immature and generic in nature. That is, the tools support the language itself, rather than its distinctively concurrent aspects. Given the widespread use of Erlang, tools developed for it point the way for tools in other actor languages and frameworks. For example, just as many Erlang tools use tracing support provided by the Erlang VM, so can other actor frameworks, e.g. Akka can use the Kamon[7] JVM monitoring system. Similarly, tools for other actor languages or frameworks could use data derived through OS-level tracing frameworks DTrace[8] and SystemTap[9] probes as we show in this section for Erlang, provided that the host language has tracing hooks into the appropriate infrastructure.

[6]http://www.erlang.org/doc/apps/observer/
[7]http://kamon.io
[8]http://dtrace.org/blogs/about/
[9]https://sourceware.org/systemtap/wiki

7.1. Refactoring for Scalability

Refactoring [Opdyke 1992; Fowler 1999; Thompson and Li 2013] is the process of changing how a program works without changing what it does. This can be done for readability, for testability, to prepare it for modification or extension, or — as is the case here — in order to improve its scalability. Because refactoring involves the transformation of source code, it is typically performed using machine support in a refactoring tool. There are a number of tools that support refactoring in Erlang: in the RELEASE project we have chosen to extend Wrangler[10] [Li et al. 2008]; other tools include Tidier [Sagonas and Avgerinos 2009] and RefactorErl [Horváth et al. 2008].

Supporting API Migration. The SD Erlang libraries modify Erlang's `global_group` library, becoming the new `s_group` library; as a result, Erlang programs using `global_group` will have to be refactored to use `s_group`. This kind of API migration problem is not uncommon, as software evolves and this often changes the API of a library. Rather than simply extend Wrangler with a refactoring to perform this particular operation, we instead added a framework for the automatic generation of API migration refactorings from a user-defined adaptor module.

Our approach to automatic API migration works in this way: when an API function's interface is changed, the author of this API function implements an *adaptor function*, defining calls to the old API in terms of the new. From this definition we automatically generate the refactoring that transforms the client code to use the new API; this refactoring can also be supplied by the API writer to clients on library upgrade, allowing users to upgrade their code automatically. The refactoring works by generating a set of rules that "fold in" the adaptation to the client code, so that the resulting code works directly with the new API. More details of the design choices underlying the work and the technicalities of the implementation can be found in a paper by Li and Thompson [2012].

Support for Introducing Parallelism. We have introduced support for parallelising explicit list operations (`map` and `foreach`), for process introduction to complete a computationally intensive task in parallel, for introducing a worker process to deal with call handling in an Erlang "generic server" and to parallelise a tail recursive function. We discuss these in turn now; more details and practical examples of the refactorings appear in a conference paper describing that work [Li and Thompson 2015].

Uses of `map` and `foreach` in list processing are among of the most obvious places where parallelism can be introduced. We have added a small library to Wrangler, called `para_lib`, which provides parallel implementations of `map` and `foreach`. The transformation from an explicit use of sequential `map/foreach` to the use of their parallel counterparts is very straightforward, even manual refactoring would not be a problem. However a `map/foreach` operation could also be implemented differently using recursive functions, list comprehensions, etc.; identifying this kind of implicit `map/foreach` usage can be done using Wrangler's code inspection facility, and a refactoring that turns an implicit `map/foreach` to an explicit `map/foreach` can also be specified using Wrangler's rule-based transformation API.

If the computations of two non-trivial tasks do not depend on each other, then they can be executed in parallel. The *Introduce a New Process* refactoring implemented in Wrangler can be used to spawn a new process to execute a task in parallel with its parent process. The result of the new process is sent back to the parent process, which will then consume it when needed. In order not to block other computations that do

[10] http://www.cs.kent.ac.uk/projects/wrangler/

not depend on the result returned by the new process, the `receive` expression is placed immediately before the point where the result is needed.

While some tail-recursive list processing functions can be refactored to an explicit map operation, many cannot due to data dependencies. For instance, an example might perform a recursion over a list while accumulating results in an accumulator variable. In such a situation it is possible to "float out" some of the computations into parallel computations. This can only be done when certain dependency constraints are satisfied, and these are done by program slicing, which is discussed below.

Support for Program Slicing. Program slicing is a general technique of program analysis for extracting the part of a program, also called the *slice*, that influences or is influenced by a given point of interest, i.e. the *slicing criterion*. Static program slicing is generally based on program dependency including both control dependency and data dependency. *Backward intra-function slicing* is used by some of the refactorings described above; it is also useful in general, and made available to end-users under Wrangler's Inspector menu [Li and Thompson 2015].

Our work can be compared with that in PaRTE[11] [Bozó et al. 2015], a tool developed in another EU project that also re-uses the Wrangler front end. This work concentrates on skeleton introduction, as does some of our work, but we go further in using static analysis and slicing in transforming programs to make them suitable for introduction of parallel structures.

7.2. Scalable Deployment

We have developed the WombatOAM tool[12] to provide a deployment, operations and maintenance framework for large-scale Erlang distributed systems. These systems typically consist of a number of Erlang nodes executing on different hosts. These hosts may have different hardware or operating systems, be physical or virtual, or run different versions of Erlang/OTP. Prior to the development of WombatOAM, deployment of systems would use scripting in Erlang and the shell, and this is the state of the art for other actor frameworks; it would be possible to adapt the WombatOAM approach to these frameworks in a straightforward way.

Architecture. The architecture of WombatOAM is summarised in Fig. 20. Originally the system had problems addressing full scalability because of the role played by the central *Master* node; in its current version an additional layer of *Middle Managers* was introduced to allow the system to scale easily to thousands of deployed nodes. As the diagram shows, the "northbound" interfaces to the web dashboard and the command-line are provided through RESTful connections to the Master. The operations of the Master are delegated to the Middle Managers that engage directly with the managed nodes. Each managed node runs a collection of *services* that collect metrics, raise alarms and so forth; we describe those now.

Services. WombatOAM is designed to collect, store and display various kinds of information and event from running Erlang systems, and these data are accessed and managed through an AJAX-based Web Dashboard; and include the following.

> *Metrics.* WombatOAM supports the collection of some hundred metrics — including, for instance, numbers of processes on a node and the message traffic in and out of the node — on a regular basis from the Erlang VMs running on each of the hosts. It can also collect metrics defined by users within other metrics collection frameworks,

[11] http://paraphrase-enlarged.elte.hu/downloads/D4-3_user_manual.pdf

[12] WombatOAM (https://www.erlang-solutions.com/products/wombat-oam.html) is a commercial tool available from Erlang Solutions Ltd.

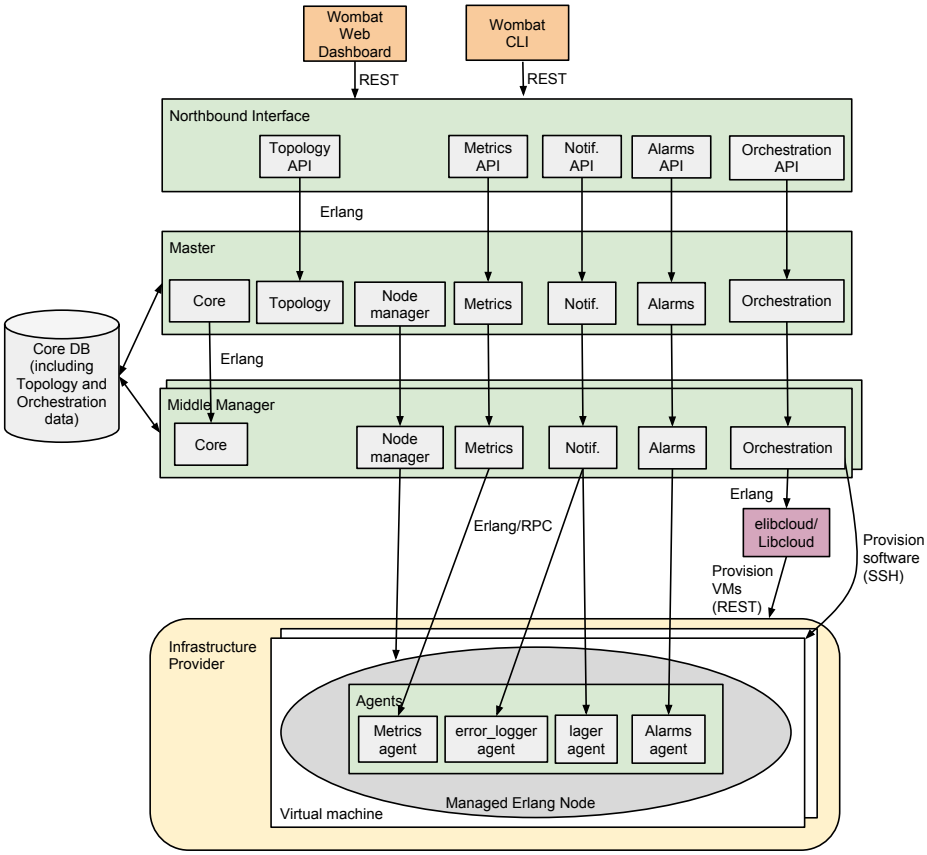

Fig. 20. The architecture of WombatOAM.

such as Folsom[13], and interface with other tools such as graphite[14] which can log and display such information. The metrics can be displayed as histograms covering different windows, such as the last fifteen minutes, hour, day, week or month.

Notifications. As well as metrics, it can support *event-based* monitoring through the collection of notifications from running nodes. Notifications, which are *one time events* can be generated using the Erlang System Architecture Support Libraries, `SASL`, which is part of the standard distribution, or the `lager` logging framework[15], and will be displayed and logged as they occur.

Alarms. Alarms are more complex entities. Alarms have a state: they can be *raised*, and once dealt with they can be *cleared*; they also have identities, so that the same alarm may be raised on the same node multiple times, and each instance will need to be dealt with separately. Alarms are generated by `SASL` or `lager`, just as for notifications.

Topology. The Topology service handles adding, deleting and discovering nodes. It also monitors whether they are accessible, and if not, it notifies the other services,

[13]Folsom collects and publishes metrics through an Erlang API: https://github.com/boundary/folsom
[14]https://graphiteapp.org
[15]https://github.com/basho/lager

and periodically tries to reconnect. When the nodes are available again, it also notifies the other services. It doesn't have a middle manager part, because it doesn't talk to the nodes directly: instead it asks the Node manager service to do so.

Node manager. This service maintains the connection to all managed nodes via the Erlang distribution protocol. If it loses the connection towards a node, it periodically tries to reconnect. It also maintains the states of the nodes in the database (e.g. if the connection towards a node is lost, the Node manager changes the node state to DOWN and raises an alarm). The Node manager doesn't have a REST API, since the node states are provided via the Topology service's REST API.

Orchestration. This service can deploy new Erlang nodes on already running machines. It can also provision new virtual machine instances using several cloud providers, and deploy Erlang nodes on those instances. For communicating with the cloud providers, the Orchestration service uses an external library called Libcloud,[16] for which Erlang Solutions has written an open source Erlang wrapper called `elibcloud`, to make Libcloud easier to use from WombatOAM. Note that WombatOAM Orchestration doesn't provide a platform for writing Erlang applications: it provides infrastructure for deploying them.

Deployment. The mechanism consists of the following five steps.

(1) *Registering a provider.* WombatOAM provides the same interface for different cloud providers which support the OpenStack standard or the Amazon EC2 API. WombatOAM also provides the same interface for using a fixed set of machines. In WombatOAM's backend, this has been implemented as two driver modules: the *elibcloud* driver module which uses the `elibcloud` and Libcloud libraries to communicate with the cloud providers, and the *SSH* driver module that keeps track of a fixed set of machines.
(2) *Uploading a release.* The release can be either a proper Erlang release archive or a set of Erlang modules. The only important aspect from WombatOAM's point of view is that start and stop commands should be explicitly specified. This will enable WombatOAM start and stop nodes when needed.
(3) *Defining a node family.* The next step is creating the node family, which is the entity that refers to a certain release, contains deployment domains that refer to certain providers, and contains other information necessary to deploy a node.
(4) *Defining a deployment domain.* At this step a deployment domain is created that specifies(i) which providers should be used for provisioning machines; (ii) the username that will be used when WombatOAM connects to the hosts using *SSH*.
(5) *Node deployment.* To deploy nodes a WombatOAM user needs only to specify the number of nodes and the node family these nodes belong to. Nodes can be dynamically added to, or removed from, the system depending on the needs of the application. The nodes are started, and WombatOAM is ready to initiate and run the application.

7.3. Monitoring and Visualisation

A key aim in designing new monitoring and visualisation tools and adapting existing ones was to provide support for systems running on parallel and distributed hardware. Specifically, in order to run modern Erlang systems, and in particular SD Erlang systems, it is necessary to understand both their single host ("multicore") and multi-host ("distributed") nature. That is, we need to be able to understand how systems run on the Erlang multicore virtual machine, where the scheduler associated with a core manages its own run queue, and processes migrate between the queues through a

[16]The unified cloud API [Apache SF 2016]: https://libcloud.apache.org

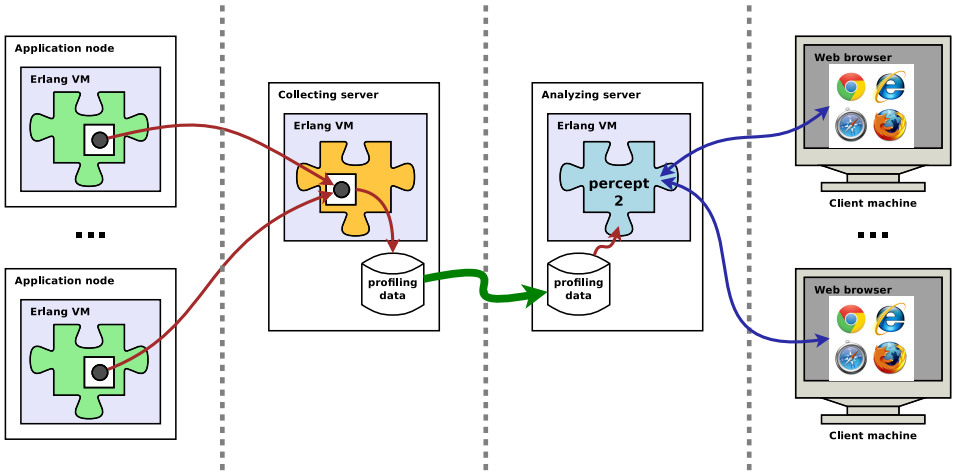

Fig. 21. Offline profiling of a distributed Erlang application using Percept2.

work stealing algorithm; at the same time, we have to understand the dynamics of a distributed Erlang program, where the user explicitly spawns processes to nodes.[17]

7.3.1. Percept2. Percept2[18] builds on the existing Percept tool to provide *post hoc* offline analysis and visualisation of Erlang systems. Percept2 is designed to allow users to visualise and tune the parallel performance of Erlang systems on a single node on a single manycore host. It visualises Erlang application level concurrency and identifies concurrency bottlenecks. Percept2 uses Erlang built in tracing and profiling to monitor process states, i.e. waiting, running, runnable, free and exiting. A waiting or suspended process is considered an inactive and a running or runnable process is considered active. As a program runs with Percept, events are collected and stored to a file. The file is then analysed, with the results stored in a RAM database, and this data is viewed through a web-based interface. The process of offline profiling for a distributed Erlang application using Percept2 is shown in Fig. 21.

Percept generates an application-level zoomable concurrency graph, showing the number of active processes at each point during profiling; dips in the graph represent low concurrency. A lifetime bar for each process is also included, showing the points during its lifetime when the process was active, as well as other per-process information.

Percept2 extends Percept in a number of ways — as detailed by Li and Thompson [2013] — including most importantly:

— Distinguishing between running and runnable time for each process: this is apparent in the process runnability comparison as shown in Fig. 22, where orange represents runnable and green represents running. This shows very clearly where potential concurrency is not being exploited.
— Showing scheduler activity: the number of active schedulers at any time during the profiling.
— Recording more information during execution, including the migration history of a process between run queues; statistics about message passing between processes:

[17]This is in contrast with the multicore VM, where programmers have no control over where processes are spawned; however, they still need to gain insight into the behaviour of their programs to tune performance.
[18]https://github.com/RefactoringTools/percept2

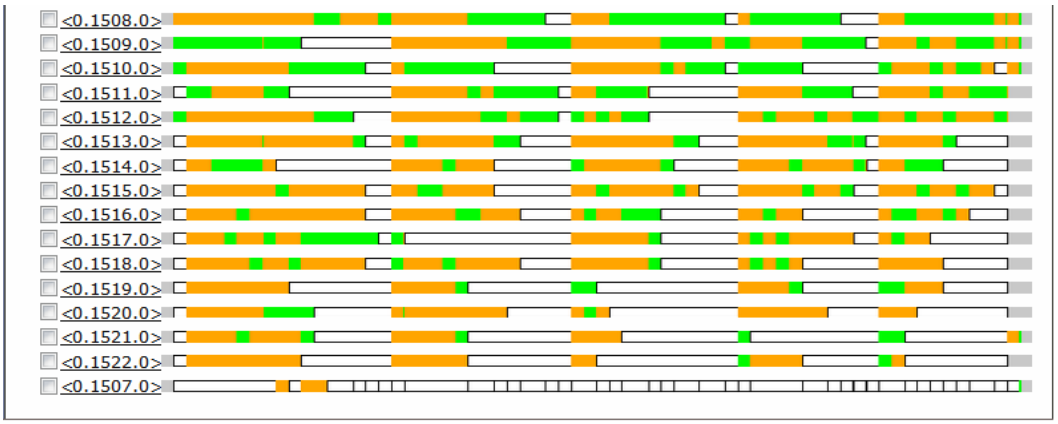

Fig. 22. Percept2: showing processes running and runnable (with 4 schedulers/run queues).

the number of messages and the average message size sent/received by a process; the accumulated runtime per-process: the accumulated time when a process is in a running state.

— Presenting the process tree: the hierarchy structure indicating the parent-child relationships between processes.
— Recording dynamic function call graph/count/time: the hierarchy structure showing the calling relationships between functions during the program run, and the amount of time spent in a function.
— Tracing of s_group activities in a distributed system. Unlike global group, s_group allows dynamic changes to the s_group structure of a distributed Erlang system. In order to support SD Erlang, we have also extended Percept2 to allow the profiling of s_group related activities, so that the dynamic changes to the s_group structure of a distributed Erlang system can be captured.

We have also improved on Percept as follows.

— Enabling finer-grained control of what is profiled. The profiling of port activities, schedulers activities, message passing, process migration, garbage collection and s_group activities can be enabled/disabled, while the profiling of process runnability (indicated by the "proc" flag) is always enabled.
— Selective function profiling of processes. In Percept2, we have built a version of fprof, which does not measure a function's own execution time, but measures everything else that fprof measures. Eliminating measurement of a function's own execution time gives users the freedom of not profiling all the function calls invoked during the program execution. For example, they can choose to profile only functions defined in their own applications' code, and not those in libraries.
— Improved dynamic function callgraph. With the dynamic function callgraph, a user is able to understand the causes of certain events, such as heavy calls of a particular function, by examining the region around the node for the function, including the path to the root of the graph. Each edge in the callgraph is annotated with the number of times the target function is called by the source function as well as further information.

Finally, we have also improved the scalability of Percept in three ways. First we have parallelised the processing of trace files so that multiple data files can be processed at the same time. We have also compressed the representation of call graphs, and cached

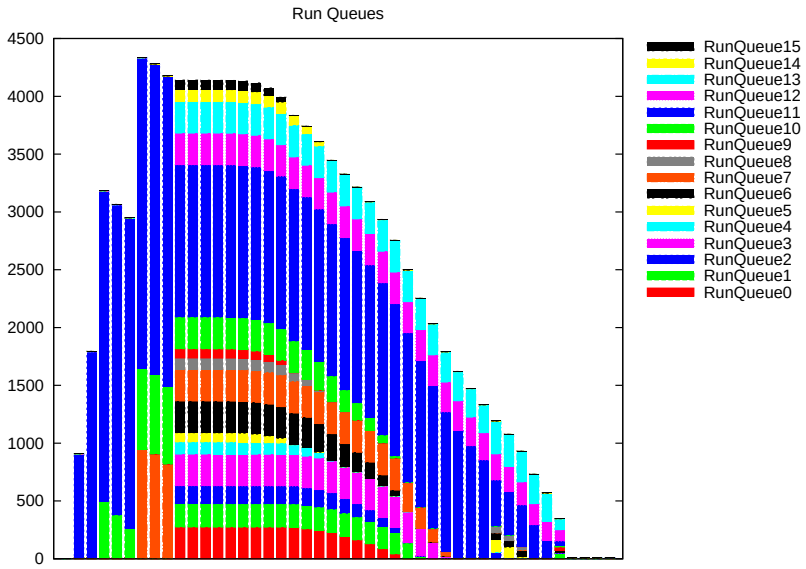

Fig. 23. DTrace: runqueue visualisation of bang benchmark with 16 schedulers (time on X-axis).

the history of generated web pages. Together these make the system more responsive and more scalable.

7.3.2. DTrace/SystemTap tracing. DTrace provides dynamic tracing support for various flavours of Unix, including BSD and Mac OS X, and SystemTap does the same for Linux; both allow the monitoring of live, running systems with minimal overhead. They can be used by administrators, language developers and application developers alike to examine the behaviour of applications, language implementations and the operating system during development or even on live production systems. In comparison to other similar tools and instrumentation frameworks, they are relatively lightweight, do not require special recompiled versions of the software to be examined, nor special post-processing tools to create meaningful information from the data gathered. Using these probs, it is possible to identify bottlenecks both in the VM itself and in applications.

VM bottlenecks are identified using a large number of probes inserted into Erlang/OTP's VM, for example to explore scheduler run-queue lengths. These probes can be used to measure the number of processes per scheduler, the number of processes moved during work stealing, the number of attempts to gain a run-queue lock, how many of these succeed immediately, etc. Figure 23 visualises the results of such monitoring; it shows how the size of run queues vary during execution of the bang BenchErl benchmark on a VM with 16 schedulers.

Application bottlenecks are identified by an alternative back-end for Percept2, based on DTrace or SystemTap instead of the Erlang built-in tracing mechanism. The implementation re-uses the existing Percept2 infrastructure as far as possible. It uses a different mechanism for collecting information about Erlang programs, a different format for the trace files, but the same storage infrastructure and presentation facilities.

7.3.3. Devo. Devo[19] [Baker et al. 2013] is designed to provide real-time online visualisation of both the low-level (single node, multiple cores) and high-level (multiple

[19]https://github.com/RefactoringTools/devo

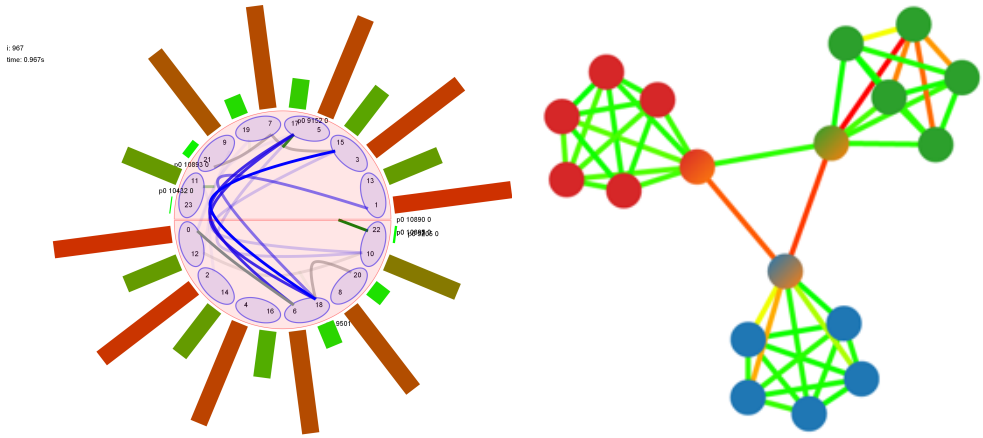

Fig. 24. Devo: low and high-level visualisations of SD-Orbit.

nodes, grouped into s_groups) aspects of Erlang and SD Erlang systems. Visualisation is within a browser, with web sockets providing the connections between the JavaScript visualisations and the running Erlang systems, instrumented through the trace tool builder (`ttb`).

Figure 24 shows visualisations from devo in both modes. On the left-hand side a single compute node is shown. This consists of two physical chips (the upper and lower halves of the diagram) with six cores each; with hyperthreading this gives twelve virtual cores, and hence 24 run queues in total. The size of these run queues is shown by both the colour and height of each column, and process migrations are illustrated by (fading) arc between the queues within the circle. A green arc shows migration on the same physical core, a grey one on the same chip, and blue shows migrations between the two chips.

On the right-hand side of Fig. 24 is a visualisation of SD-Orbit in action. Each node in the graph represents an Erlang node, and colours (red, green, blue and orange) are used to represent the s_groups to which a node belongs. As is evident, the three nodes in the central triangle belong to multiple groups, and act as routing nodes between the other nodes. The colour of the arc joining two nodes represents the current intensity of communication between the nodes (green quiescent; red busiest).

7.3.4. SD-Mon. SD-Mon is a tool specifically designed for monitoring SD-Erlang systems. This purpose is accomplished by means of a *shadow network* of agents, that collect data from a running system. An example deployment is shown in Fig. 25, where blue dots represent nodes in the target system and the other nodes make up the SD-Mon infrastructure. The network is deployed on the basis of a configuration file describing the network architecture in terms of hosts, Erlang nodes, global group and s_group partitions. Tracing to be performed on monitored nodes is also specified within the configuration file.

An agent is started by a master SD-Mon node for each s_group and for each free node. Configured tracing is applied on every monitored node, and traces are stored in binary format in the agent file system. The shadow network follows system changes so that agents are started and stopped at runtime as required, as shown in Fig. 26. Such changes are persistently stored so that the last configuration can be reproduced after a restart. Of course, the shadow network can be always updated via the User Interface.

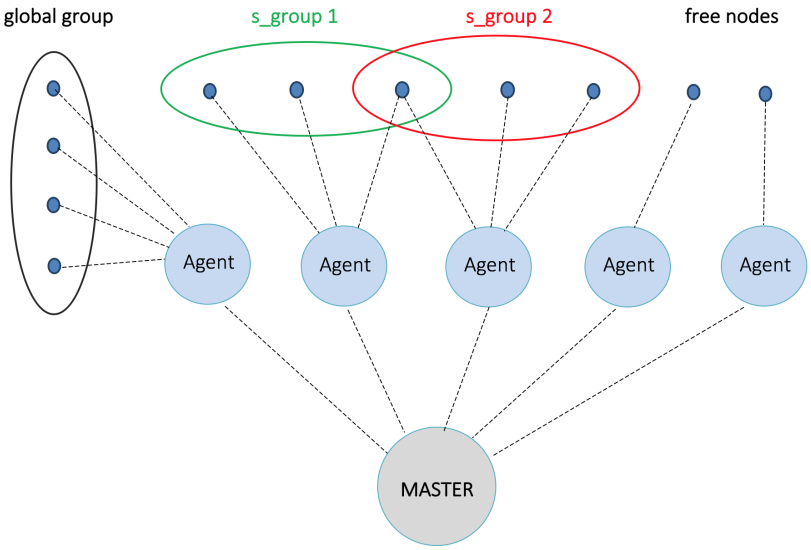

Fig. 25. SD-Mon architecture.

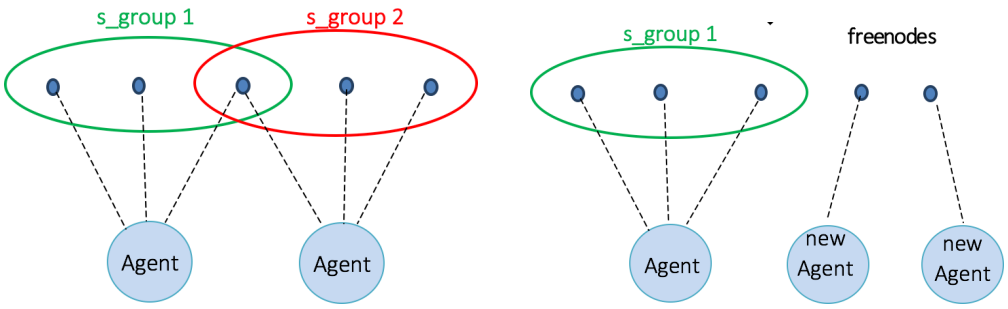

Fig. 26. SD-Mon evolution. Before (left) and after eliminating s_group 2 (right).

Each agent takes care of an s_group or of a free node. At start-up it tries to get in contact with its nodes and apply the tracing to them as stated by the master. Binary files are stored in the host file system. Tracing is internally used in order to track s_group operations happening at runtime. An asynchronous message is sent to the master whenever one of these changes occurs. Since each process can only be traced by a single process at a time, each node (included those belonging to more than one s_group) is controlled by only one agent. When a node is removed from a group or when the group is deleted, another agent takes over, as shown in Fig. 26. When an agent is stopped, all traces on the controlled nodes are switched off.

The monitoring network is also supervised, in order to take account of network fragility, and when an agent node goes down another node is deployed to play its role; there are also periodic consistency checks for the system as a whole, and when an inconsistency is detected then that part of the system can be restarted.

SD-Mon does more than monitor activities one node at a time. In particular inter-node and inter-group messages are displayed at runtime. As soon as an agent is

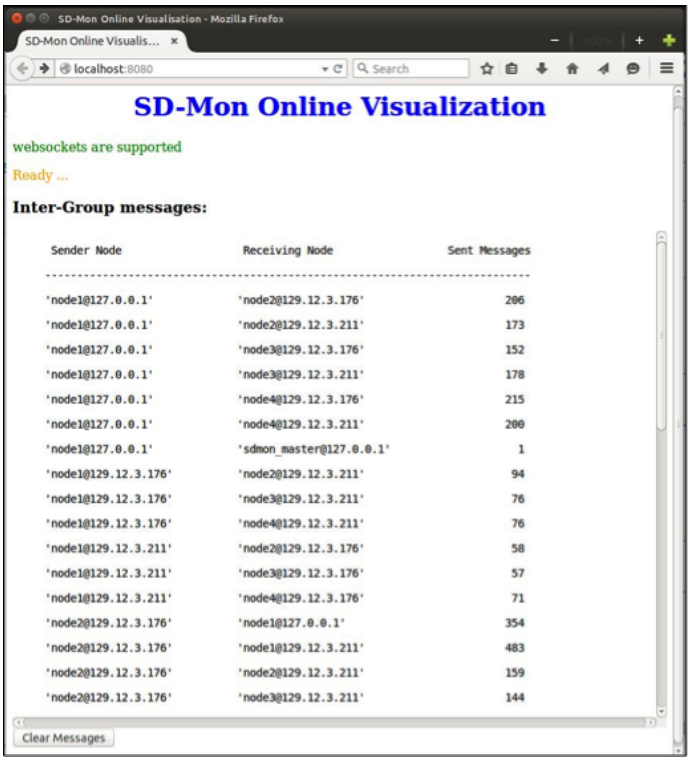

Fig. 27. SD-Mon: online monitoring.

stopped, the related tracing files are fetched across the network by the master and they are made available in a readable format in the master file system.

SD-Mon provides facilities for online visualisation of this data, as well as *post hoc* offline analysis. Figure 27 shows, in real time, messages that are sent *between* s_groups. This data can also be used as input to the animated Devo visualisation, as illustrated in the right-hand side of Fig. 24.

8. SYSTEMIC EVALUATION

Preceding sections have investigated the improvements of individual aspects of an Erlang system, e.g. ETS tables in Section 6.1. This section analyses the impact of the new tools and technologies from Sections 5 to 7 in concert. We do so by investigating the deployment, reliability, and scalability of the Orbit and ACO benchmarks from Section 3. The experiments reported here are representative. Similar experiments show consistent results for a range of micro-benchmarks, several benchmarks, and the very substantial (approximately 150K lines of Erlang) Sim-Diasca case study [Boudeville 2012] on several state of the art NUMA architectures, and the four clusters specified in Appendix A. A coherent presentation of many of these results is available in an article by Chechina et al. [2017] and in a RELEASE project deliverable[20]. The bulk of the experiments reported here are conducted on the Athos cluster using Erlang/OTP 17.4 and the associated SD Erlang libraries.

[20]See Deliverable 6.2, available online. http://www.release-project.eu/documents/D6.2.pdf

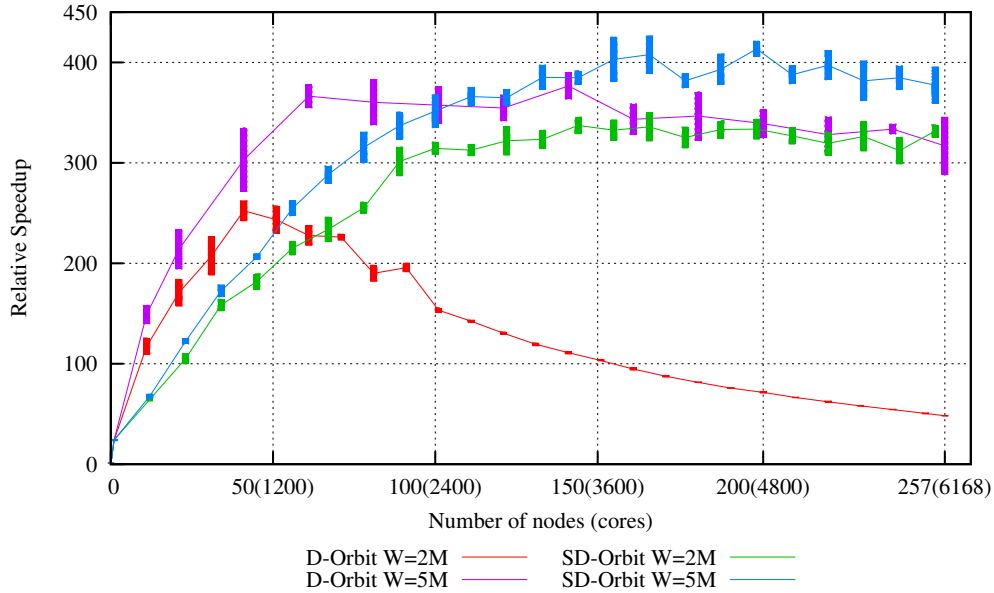

Fig. 28. Speedup of distributed Erlang vs. SD Erlang Orbit (2M and 5M elements) [Strong Scaling].

The experiments cover two measures of scalability. As Orbit does a fixed size computation, the scaling measure is relative speedup (or strong scaling), i.e. speedup relative to execution time on a single core. As the work in ACO increases with compute resources, weak scaling [21] is the appropriate measure. The benchmarks also evaluate different aspects of s_groups: Orbit evaluates the scalability impacts of network connections, while ACO evaluates the impact of both network connections and the global namespace required for reliability.

8.1. Orbit

Figure 28 shows the speedup of the D-Orbit and SD-Orbit benchmarks from Section 5.1.1. The measurements are repeated seven times, and we plot standard deviation. Results show that D-Orbit performs better on a small number of nodes as communication is direct, rather than via a gateway node. As the number of nodes grows, however, SD-Orbit delivers better speedups, i.e. beyond 80 nodes in case of 2M orbit elements, and beyond 100 nodes in case of 5M orbit elements. When we increase the size of Orbit beyond 5M, the D-Orbit version fails due to the fact that some VMs exceed the available RAM of 64GB. In contrast SD-Orbit experiments run successfully even for an orbit with 60M elements.

8.2. Ant Colony Optimisation (ACO)

Deployment. The deployment and monitoring of ACO, and of the large (150K lines of Erlang) Sim-Diasca simulation using WombatOAM (Section 7.2) on the Athos cluster is detailed in [Chechina et al. 2017].

An example experiment deploys 10,000 Erlang nodes without enabling monitoring, and hence allocates three nodes per core (i.e. 72 nodes on each of 139 24-core Athos

[21] Weak scaling measures how runtime varies with the number of cores with a fixed problem size per core: a constant runtime is an ideal result.

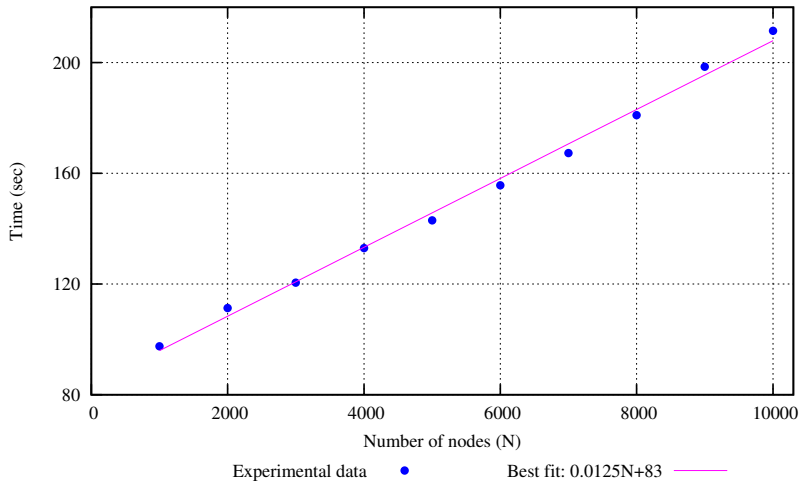

Fig. 29. Wombat Deployment Time

hosts). Figure 29 shows that WombatOAM deploys the nodes in 212s, which is approximately 47 nodes per second. It is more common to have at least one core per Erlang node, and in a related experiment 5000 nodes are deployed, one per core (i.e. 24 nodes on each of 209 24-core Athos hosts) in 101s, or 50 nodes per second. Crucially in both cases the deployment time is linear in the number of nodes. The deployment time could be reduced to be logarithmic in the number of nodes using standard divide-and-conquer techniques. However there hasn't been the demand to do so as most Erlang systems are long running servers.

The measurement data shows two important facts: it shows that WombatOAM scales well (up to a deployment base of 10,000 Erlang nodes), and that WombatOAM is non-intrusive because its overhead on a monitored node is typically less than 1.5% of effort on the node.

We conclude that WombatOAM is capable of deploying and monitoring substantial distributed Erlang and SD Erlang programs. The experiments in the remainder of this section use standard distributed Erlang configuration file deployment.

Reliability. SD Erlang changes the organisation of processes and their recovery data at the language level, so we seek to show that these changes have not disrupted Erlang's world-class reliability mechanisms at this level. As we haven't changed them we don't exercise Erlang's other reliability mechanisms, e.g. those for managing node failures, network congestion, etc. A more detailed study of SD Erlang reliability, including the use of replicated databases for recovering Instant Messenger chat sessions, finds similar results [Chechina et al. 2016].

We evaluate the reliability of two ACO versions using a Chaos Monkey service that kills processes in the running system at random [Bennett and Tseitlin 2012]. Recall that GR-ACO provides reliability by registering the names of critical processes globally, and SR-ACO registers them only within an s_group (Section 3.2).

For both GR-ACO and SR-ACO a Chaos Monkey runs on every Erlang node, i.e. master, submasters, and colony nodes, killing a random Erlang process every second. Both ACO versions run to completion. Recovery, at this failure frequency, has no measurable impact on runtime. This is because processes are recovered within the Virtual machine using (globally synchronised) local recovery information. For example,

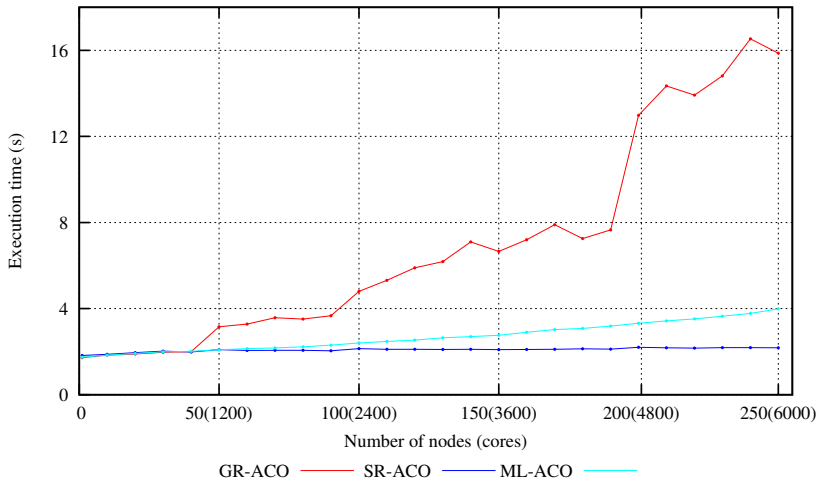

Fig. 30. ACO execution times, Erlang/OTP 17.4 (RELEASE) [Weak Scaling].

on a common X86/Ubuntu platform typical Erlang process recovery times are around 0.3ms, so around 1000x less than the Unix process recovery time on the same platform [Lutac et al. 2016]. We have conducted more detailed experiments on an Instant Messenger benchmark, and obtained similar results [Chechina et al. 2016].

We conclude that both GR-ACO and SR-ACO are reliable, and that SD Erlang preserves distributed Erlang reliability model. The remainder of the section outlines the impact of maintaining the recovery information required for reliability on scalability.

Scalability. Figure 30 compares the runtimes of the ML, GR, and SR versions of ACO (Section 3.2) on Erlang/OTP 17.4(RELEASE). As outlined in Section 5.1.1, GR-ACO not only maintains a fully connected graph of nodes, it registers process names for reliability, and hence scales significantly worse than the unreliable ML-ACO. We conclude that providing reliability with standard distributed Erlang process registration dramatically limits scalability.

While ML-ACO does not provide reliability, and hence doesn't register process names, it maintains a fully connected graph of nodes which limits scalability. SR-ACO, that maintains connections and registers process names only within s_groups scales best of all. Figure 30 illustrates how maintaining the process namespace, and fully connected network, impacts performance. This reinforces the evidence from the Orbit benchmarks, and others, that partitioning the network of Erlang nodes significantly improves performance at large scale.

To investigate the impact of SD Erlang on network traffic, we measure the number of sent and received packets on the GPG cluster for three versions of ACO: ML-ACO, GR-ACO, and SR-ACO. Figure 31 shows the total number of sent packets. The highest traffic (the red line) belongs to the GR-ACO and the lowest traffic belongs to the SR-ACO (dark blue line). This shows that SD Erlang significantly reduces the network traffic between Erlang nodes. Even with the s_group name registration SR-ACO has less network traffic than ML-ACO that has no global name registration.

8.3. Evaluation Summary

We have shown that WombatOAM is capable of deploying and monitoring substantial distributed Erlang and SD Erlang programs like ACO and Sim-Diasca. The

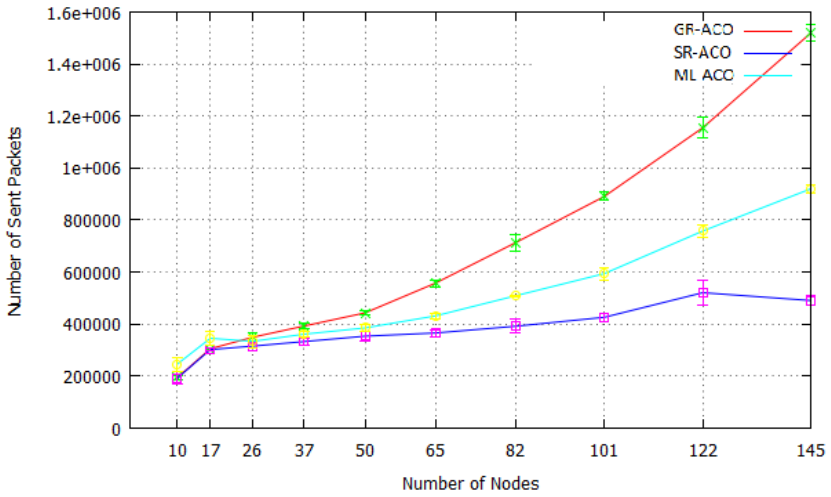

Fig. 31. Number of sent packets in ML-ACO, GR-ACO, and SR-ACO.

Chaos Monkey experiments with GR-ACO and SR-ACO show that both are reliable, and hence that SD Erlang preserves the distributed Erlang language-level reliability model.

As SD Orbit scales better than D-Orbit, SR-ACO scales better than ML-ACO, and SR-ACO has significantly less network traffic, we conclude that, even when global recovery data is not maintained, partitioning the fully-connected network into s_groups reduces network traffic and improves performance. While the distributed Orbit instances (W=2M) and (W=5M) reach scalability limits at around 40 and 60 nodes, Orbit scales to 150 nodes on SD Erlang (limited by input size), and SR-ACO is still scaling well on 256 nodes (6144 cores). Hence not only have we exceeded the 60 node scaling limits of distributed Erlang identified in Section 4.2, we have not reached the scaling limits of SD Erlang on this architecture.

Comparing GR-ACO and ML-ACO scalability curves shows that maintaining global recovery data, i.e. a process name space, dramatically limits scalability. Comparing GR-ACO and SR-ACO scalability curves shows that scalability can be recovered by partitioning the nodes into appropriately-sized s_groups, and hence maintaining the recovery data only within a relatively small group of nodes. These results are consistent with other experiments.

9. DISCUSSION

Distributed actor platforms like Erlang, or Scala with Akka, are a common choice for internet-scale system architects as the model, with its automatic, and VM-supported reliability mechanisms makes it *extremely easy to engineer scalable reliable systems*. Targeting emergent server architectures with hundreds of hosts and tens of thousands of cores, we report a systematic effort to improve the scalability of a leading distributed actor language, while preserving reliability. The work is a *vade mecum* for addressing scalability of reliable actor languages and frameworks. It is also high impact, with downloads of our improved Erlang/OTP running at 50K a month.

We have undertaken the first systematic study of scalability in a distributed actor language, covering VM, language and persistent storage levels. We have developed the BenchErl and DE-Bench tools for this purpose. Key VM-level scalability issues we identify include contention for shared ETS tables and for commonly-used shared

resources like timers. Key language scalability issues are the costs of maintaining a fully-connected network, maintaining global recovery information, and explicit process placement. Unsurprisingly the scaling issues for this distributed actor language are common to other distributed or parallel languages and frameworks with other paradigms like CHARM++ [Kale and Krishnan 1993], Cilk [Blumofe et al. 1995], or Legion [Grimshaw et al. 1997]. We establish scientifically the folklore limitations of around 60 connected nodes for distributed Erlang (Section 4).

The actor model is no panacea, and there can still be scalability problems in the algorithms that we write, either within a single actor or in the way that we structure communicating actors. A range of pragmatic issues also impact the performance and scalability of actor systems, including memory occupied by processes (even when quiescent), mailboxes filling up, etc. Identifying and resolving these problems is where tools like Percept2 and WombatOAM are needed. However, *many of the scalability issues arise where Erlang departs from the private state principle of the actor model*, e.g. in maintaining shared state in ETS tables, or a shared global process namespace for recovery.

We have designed and implemented a set of Scalable Distributed (SD) Erlang libraries to address *language-level* scalability issues. The key constructs are s_groups for partitioning the network and global process namespace, and semi-explicit process placement for deploying distributed Erlang applications on large heterogeneous architectures in a portable way. We have provided a state transition operational semantics for the new s_groups, and validated the library implementation against the semantics using QuickCheck (Section 5).

To improve the scalability of the Erlang VM and libraries we have improved the implementation of shared ETS tables, time management and load balancing between schedulers. Following a systematic analysis of ETS tables, the number of fine-grained (bucket) locks and of reader groups have been increased. We have developed and evaluated four new techniques for improving ETS scalability:(i) programmer control of number of bucket locks; (ii) a contention-adapting tree data structure for `ordered_sets`; (iii) queue delegation locking; and (iv) eliminating the locks in the meta table. We have introduced a new scheduler utilisation balancing mechanism to spread work to multiple schedulers (and hence cores), and new synchronisation mechanisms to reduce contention on the widely-used time management mechanisms. By June 2015, with Erlang/OTP 18.0, the majority of these changes had been included in the primary releases. In any scalable actor language implementation such thoughtful design and engineering will be required to schedule large numbers of actors on hosts with many cores, and to minimise contention on shared VM resources (Section 6).

To facilitate the development of large Erlang systems, and to make them understandable we have developed a range of tools. The proprietary WombatOAM tool deploys and monitors large distributed Erlang systems over multiple, and possibly heterogeneous, clusters or clouds. We have made open source releases of four concurrency tools: Percept2 now detects concurrency bad smells; Wrangler provides enhanced concurrency refactoring; the Devo tool is enhanced to provide interactive visualisation of SD Erlang systems; and the new SD-Mon tool monitors SD Erlang systems. We anticipate that these tools will guide the design of tools for other large scale distributed actor languages and frameworks (Section 7).

We report on the reliability and scalability implications of our new technologies using a range of benchmarks, and consistently use the Orbit and ACO benchmarks throughout the article. While we report measurements on a range of NUMA and cluster architectures, the key scalability experiments are conducted on the Athos cluster with 256 hosts (6144 cores). Even when global recovery data is not maintained, partitioning the network into s_groups reduces network traffic and improves the perfor-

Table II. Cluster Specifications (RAM per host in GB).

		Cores					
Name	**Hosts**	**per host**	**total**	**max avail.**	**Processor**	**RAM**	**Inter-connection**
GPG	20	16	320	320	Intel Xeon E5-2640v2 8C, 2GHz	64	10GB Ethernet
Kalkyl	348	8	2,784	1,408	Intel Xeon 5520v2 4C, 2.26GHz	24–72	InfiniBand 20 Gb/s
TinTin	160	16	2,560	2,240	AMD Opteron 6220v2 Bulldozer 8C, 3.0GHz	64–128	2:1 oversubscribed QDR Infiniband
Athos	776	24	18,624	6,144	Intel Xeon E5-2697v2 12C, 2.7GHz	64	Infiniband FDR14

mance of the Orbit and ACO benchmarks above 80 hosts. Crucially we exceed the 60 node limit for distributed Erlang and do not reach the scalability limits of SD Erlang with 256 nodes/VMs and 6144 cores. Chaos Monkey experiments show that two versions of ACO are reliable, and hence that SD Erlang preserves the Erlang reliability model. However the ACO results show that maintaining global recovery data, i.e. a global process name space, dramatically limits scalability in distributed Erlang. Scalability can, however, be recovered by maintaining recovery data only within appropriately sized s_groups. These results are consistent with experiments with other benchmarks and on other architectures (Section 8).

In future work we plan to incorporate RELEASE technologies, along with other technologies in a generic framework for building performant large scale servers. In addition, preliminary investigations suggest that some SD Erlang ideas could improve the scalability of other actor languages.[22] For example the Akka framework for Scala could benefit from semi-explicit placement, and Cloud Haskell from partitioning the network.

Appendix A: Architecture Specifications

The specifications of the clusters used for measurement are summarised in Table II. We also use the following NUMA machines. (1) An AMD Bulldozer with 16M L2/16M L3 cache, 128GB RAM, four AMD Opteron 6276s at 2.3 GHz, 16 "Bulldozer" cores each, giving a total of 64 cores. (2) An Intel NUMA with 128GB RAM, four Intel Xeon E5-4650s at 2.70GHz, each with eight hyperthreaded cores, giving a total of 64 cores.

ACKNOWLEDGMENTS

We would like to thank the entire RELEASE project team for technical insights and administrative support. Roberto Aloi and Enrique Fernandez Casado contributed to the development and measurement of WombatOAM. This work has been supported by the European Union grant RII3-CT-2005-026133 "SCIEnce: Symbolic Computing Infrastructure in Europe", IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software", and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 "HPC-GAP: High Performance Computational Algebra and Discrete Mathematics".

REFERENCES

Gul Agha. 1985. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Ph.D. Dissertation. MIT.

Gul Agha. 1986. An Overview of Actor Languages. *SIGPLAN Not.* 21, 10 (1986), 58–67.

AMD 2015. Bulldozer (microarchitecture). (2015). https://en.wikipedia.org/wiki/Bulldozer_(microarchitecture).

Apache SF. 2016. Liblcoud. (2016). https://libcloud.apache.org/.

[22]See Deliverable 6.7 (http://www.release-project.eu/documents/D6.7.pdf), available online.

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

Joe Armstrong. 2007. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.

Joe Armstrong. 2010. Erlang. *Commun. ACM* 53 (2010), 68–75. Issue 9.

Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. 2012. A Scalability Benchmark Suite for Erlang/OTP. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang*, Torben Hoffman and John Hughes (Eds.). ACM, New York, NY, USA, 33–42. DOI:http://dx.doi.org/10.1145/2364489.2364495

Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. ACM, New York, NY, USA, 2–10.

Robert Baker, Peter Rodgers, Simon Thompson, and Huiqing Li. 2013. Multi-level Visualization of Concurrent and Distributed Computation in Erlang. In *Visual Languages and Computing (VLC): The 19th International Conference on Distributed Multimedia Systems (DMS 2013)*.

Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer* (2nd ed.). Morgan and Claypool.

Basho Technologies. 2014. Riakdocs. Basho Bench. (2014). http://docs.basho.com/riak/latest/ops/building/benchmarking/.

J. E. Beasley. 1990. OR-Library: Distributing Test Problems by Electronic Mail. *The Journal of the Operational Research Society* 41, 11 (1990), 1069–1072. http://www.jstor.org/stable/2582903 Datasets available at http://people.brunel.ac.uk/~mastjjb/jeb/orlib/wtinfo.html.

Cory Bennett and Ariel Tseitlin. 2012. Chaos Monkey Released into the Wild. Netflix Blog. (2012).

Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 207–216.

Olivier Boudeville. 2012. Technical Manual of the Sim-Diasca Simulation Engine. *EDF R&D* (2012).

István Bozó, Viktória Fördős, Dániel Horpácsi, Zoltán Horváth, Tamás Kozsik, Judit Kőszegi, and Melinda Tóth. 2015. Refactorings to Enable Parallelization. In *Trends in Functional Programming: 15th International Symposium, TFP 2014. Revised Selected Papers (LNCS)*, Jurriaan Hage and Jay McCarthy (Eds.), Vol. 8843. Springer, 104–121.

Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, USA, 157–166. DOI:http://dx.doi.org/10.1145/2442516.2442532

Francesco Cesarini and Simon Thompson. 2009. *Erlang Programming: A Concurrent Approach to Software Development* (1st ed.). O'Reilly Media.

Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. 2001. *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco, CA, USA.

Natalia Chechina, Huiqing Li, Amir Ghaffari, Simon Thompson, and Phil Trinder. 2016. Improving the Network Scalability of Erlang. *J. Parallel Distrib. Comput.* 90, C (2016), 22–34. DOI:http://dx.doi.org/10.1016/j.jpdc.2016.01.002

Natalia Chechina, Kenneth MacKenzie, Simon Thompson, Phil Trinder, Olivier Boudeville, Viktória Fördős, Csaba Hoch, Amir Ghaffari, and Mario Moro Hernandez. 2017. Evaluating Scalable Distributed Erlang for Scalability and Reliability. *IEEE Transactions on Parallel and Distributed Systems* (2017). DOI:http://dx.doi.org/10.1109/TPDS.2017.2654246

Natalia Chechina, Mario Moro Hernandez, and Phil Trinder. 2016. A Scalable Reliable Instant Messenger using the SD Erlang Libraries. In *Erlang'16*. ACM, New York, NY, USA, 33–41.

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, NY, USA, 268–279.

H. A. J. Crauwels, C. N. Potts, and L. N. van Wassenhove. 1998. Local Search Heuristics for the Single Machine Total Weighted Tardiness Scheduling Problem. *INFORMS Journal on Computing* 10, 3 (1998), 341–350. DOI:http://dx.doi.org/10.1287/ijoc.10.3.341 The datasets from this paper are included in Beasley's ORLIB.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

David Dewolfs, Jan Broeckhove, Vaidy Sunderam, and Graham E. Fagg. 2006. FT-MPI, Fault-Tolerant Metacomputing and Generic Name Services: A Case Study. In *EuroPVM/MPI'06*. Springer-Verlag, Berlin, Heidelberg, 133–140.

Alan AA Donovan and Brian W Kernighan. 2015. *The Go programming language*. Addison-Wesley Professional.

Marco Dorigo and Thomas Stützle. 2004. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA.

Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. 2007. SNZI: scalable NonZero indicators. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, NY, USA, 13–22. DOI:http://dx.doi.org/10.1145/1281100.1281106

Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Haskell '11*. ACM, 118–129.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Ana Gainaru and Franck Cappello. 2015. Errors and Faults. In *Fault-Tolerance Techniques for High-Performance Computing*. Springer International Publishing, 89–144.

Martin Josef Geiger. 2010. *New Instances for the Single Machine Total Weighted Tardiness Problem*. Technical Report Research Report 10-03-01. Datasets available at http://logistik.hsu-hh.de/SMTWTP.

Guillaume Germain. 2006. Concurrency Oriented Programming in Termite Scheme. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. ACM, New York, NY, USA, 20–20.

Amir Ghaffari. 2014a. DE-Bench, A Benchmark Tool for Distributed Erlang. (2014). https://github.com/amirghaffari/DEbench.

Amir Ghaffari. 2014b. Investigating the Scalability Limits of Distributed Erlang. In *Proceedings of the 13th ACM SIGPLAN Workshop on Erlang*. ACM, 43–49.

Amir Ghaffari, Natalia Chechina, Phip Trinder, and Jon Meredith. 2013. Scalable Persistent Storage for Erlang: Theory and Practice. In *Proceedings of the 12th ACM SIGPLAN Workshop on Erlang*. ACM, New York, NY, USA, 73–74.

Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33 (2002), 51–59. Issue 2.

Andrew S Grimshaw, Wm A Wulf, and the Legion team. 1997. The Legion vision of a worldwide virtual computer. *Commun. ACM* 40, 1 (1997), 39–45.

Carl Hewitt. 2010. Actor Model for Discretionary, Adaptive Concurrency. *CoRR* abs/1008.1459 (2010).

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI'73*. Morgan Kaufmann, San Francisco, CA, USA, 235–245.

Rich Hickey. 2008. The Clojure Programming Language. In *DLS'08*. ACM, New York, NY, USA, 1:1–1:1.

Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. 2008. Building a refactoring tool for Erlang. In *Workshop on Advanced Software Development Tools and Techniques, WASDETT 2008*.

Laxmikant V Kale and Sanjeev Krishnan. 1993. CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, Vol. 28. ACM, 91–108.

David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. 2013. On the Scalability of the Erlang Term Storage. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*. ACM, New York, NY, USA, 15–26. http://doi.acm.org/10.1145/2505305.2505308

David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. 2014. Delegation Locking Libraries for Improved Performance of Multithreaded Programs. In *Euro-Par 2014 Parallel Processing. (LNCS)*, Vol. 8632. Springer, 572–583. DOI:http://dx.doi.org/10.1007/978-3-319-09873-9_48

David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. 2017. Queue Delegation Locking. *IEEE Transactions on Parallel and Distributed Systems* (2017). To appear.

Rusty Klophaus. 2010. Riak Core: Building Distributed Applications Without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10)*. ACM, New York, NY, USA, Article 14, 1 pages. DOI:http://dx.doi.org/10.1145/1900160.1900176

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. DOI:http://dx.doi.org/10.1145/1773912.1773922

J. Lee and others. 2010. Python Actor Runtime Library. (2010). osl.cs.uiui.edu/parley/.

Huiqing Li and Simon Thompson. 2012. Automated API Migration in a User-Extensible Refactoring Tool for Erlang Programs. In *Automated Software Engineering, ASE'12*, Tim Menzies and Motoshi Saeki (Eds.). IEEE Computer Society.

Huiqing Li and Simon Thompson. 2013. Multicore profiling for Erlang programs using Percept2. In *Proceedings of the Twelfth ACM SIGPLAN workshop on Erlang*.

Huiqing Li and Simon Thompson. 2014. Improved Semantics and Implementation Through Property-Based Testing with QuickCheck. In *9th International Workshop on Automation of Software Test*.

Huiqing Li and Simon Thompson. 2015. Safe Concurrency Introduction through Slicing. In *PEPM '15*. ACM SIGPLAN.

Huiqing Li, Simon Thompson, György Orosz, and Melinda Töth. 2008. Refactoring with Wrangler, updated. In *ACM SIGPLAN Erlang Workshop*, Vol. 2008.

Frank Lubeck and Max Neunhoffer. 2001. Enumerating Large Orbits and Direct Condensation. *Experimental Mathematics* 10, 2 (2001), 197–205.

Andreea Lutac, Natalia Chechina, Gerardo Aragon-Camarasa, and Phil Trinder. 2016. Towards Reliable and Scalable Robot Communication. In *Proceedings of the 15th International Workshop on Erlang (Erlang 2016)*. ACM, New York, NY, USA, 12–23. http://doi.acm.org/10.1145/2975969.2975971

LWN.net. 2006. The high-resolution timer API. (Jan. 2006). https://lwn.net/Articles/167897/

Kenneth MacKenzie, Natalia Chechina, and Phil Trinder. 2015. Performance Portability through Semi-explicit Placement in Distributed Erlang. In *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang*. ACM, 27–38.

Jeff Matocha and Tracy Camp. 1998. A Taxonomy of Distributed Termination Detection Algorithms. *Journal of Systems and Software* 43, 221 (1998), 207–221.

Nicholas D Matsakis and Felix S Klock II. 2014. The Rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

Robert McNaughton. 1959. Scheduling with Deadlines and Loss Functions. *Management Science* 6, 1 (1959), 1–12.

Martin Odersky and others. 2012. The Scala Programming Language. (2012). www.scala-lang.org.

William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Nikolaos Papaspyrou and Konstantinos Sagonas. 2012. On Preserving Term Sharing in the Erlang Virtual Machine. In *Proceedings of the 11th ACM SIGPLAN Erlang Workshop*, Torben Hoffman and John Hughes (Eds.). ACM, 11–20. DOI:http://dx.doi.org/10.1145/2364489.2364493

RELEASE Project Team. 2015. EU Framework 7 Project 287510 (2011–2015). (2015). http://www.release-project.eu.

Konstantinos Sagonas and Thanassis Avgerinos. 2009. Automatic refactoring of Erlang programs. In *Principles and Practice of Declarative Programming (PPDP'09)*. ACM, 13–24.

Konstantinos Sagonas and Kjell Winblad. 2014. More Scalable Ordered Set for ETS Using Adaptation. In *Proc. ACM SIGPLAN Workshop on Erlang*. ACM, 3–11. DOI:http://dx.doi.org/10.1145/2633448.2633455

Konstantinos Sagonas and Kjell Winblad. 2015. Contention Adapting Search Trees. In *Proceedings of the 14th International Symposium on Parallel and Distributed Computing*. IEEE Computing Society, 215–224. DOI:http://dx.doi.org/10.1109/ISPDC.2015.32

Konstantinos Sagonas and Kjell Winblad. 2016. Efficient Support for Range Queries and Range Updates Using Contention Adapting Search Trees. In *Languages and Compilers for Parallel Computing - 28th International Workshop (LNCS)*, Vol. 9519. Springer, 37–53. DOI:http://dx.doi.org/10.1007/978-3-319-29778-1_3

Marc Snir, Steve W. Otto, D. W. Walker, Jack Dongarra, and Steven Huss-Lederman. 1995. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA.

Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-Typed Actors for Java. In *ECOOP'08*. Springer-Verlag, Berlin, Heidelberg, 104–128.

Don Syme, Adam Granicz, and Antonio Cisternino. 2015. *Expert F# 4.0*. Springer.

Simon Thompson and Huiqing Li. 2013. Refactoring tools for functional languages. *Journal of Functional Programming* 23, 3 (2013).

Marcus Völker. 2014. Linux Timers. (May 2014). https://upvoid.com/devblog/2014/05/linux-timers/

WhatsApp. 2015. (2015). https://www.whatsapp.com/

Tom White. 2012. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated)*. O'Reilly. I–XXIII, 1–657 pages.

Ulf Wiger. 2000. Industrial-Strength Functional programming: Experiences with the Ericsson AXD301 Project. In *Implementing Functional Languages (IFL'00)*. Springer-Verlag, Aachen, Germany.

Received December 2015; revised September 2016; accepted Sometime 2017