A Scalability Benchmark Suite for Erlang/OTP

Stavros Aronis [1] Nikolaos Papaspyrou [2] Katerina Roukounaki [2] Konstantinos Sagonas [1,2]
Yiannis Tsiouris [2] Ioannis E. Venetis [2]

[1] Department of Information Technology, Uppsala University, Sweden
[2] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
release@softlab.ntua.gr

Abstract

Programming language implementers rely heavily on benchmarking for measuring and understanding performance of algorithms, architectural designs, and trade-offs between alternative implementations of compilers, runtime systems, and virtual machine components. Given this fact, it seems a bit ironic that it is often more difficult to come up with a good benchmark suite than a good implementation of a programming language.

This paper presents the main aspects of the design and the current status of bencherl, a publicly available scalability benchmark suite for applications written in Erlang. In contrast to other benchmark suites, which are usually designed to report a particular performance point, our benchmark suite aims to assess *scalability*, i.e., help developers to study a set of performance points that show how an application's performance changes when additional resources (e.g., CPU cores, schedulers, etc.) are added. We describe the scalability dimensions that the suite aims to examine and present its infrastructure and current set of benchmarks. We also report some limited set of performance results in order to show the capabilities of our suite.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed and parallel languages; Applicative (functional) languages; D.1.3 [*Software*]: Concurrent Programming—Parallel programming; C.4 [*Performance of Systems*]: Measurement techniques

General Terms Experimentation, Measurement, Performance

Keywords benchmarking, scalability, virtual machine interpreters, multicore, Erlang

1. Introduction

Concurrent applications in Erlang typically spawn large numbers of processes which by default have no shared memory and communicate with message passing. Processes may be created on remote nodes too; communication with remote processes is transparent in the sense that it works exactly as communication with local processes.

The improvement of hardware and network technologies makes it possible to (a) create Erlang nodes that host more processes, as nodes typically run on systems with many multicore processors and large amounts of memory, and (b) distribute computation easily over multiple remote nodes, as the overhead of communication between them is reduced. We would therefore expect that concurrent applications in Erlang can *benefit immediately and immensely* from the progress of computer technology, especially with the direction that this progress clearly has taken.

Alas, things are not that simple. Experience with Erlang shows that some applications do not scale so well for various reasons, known or unknown. Since 2006, the Erlang/OTP system from Ericsson has been extended to support Symmetric MultiProcessing (SMP) in the form that is commonly found these days in multicore machines. A primary goal was to maintain stability while giving incremental performance improvements in each release. This approach has proved to work well for the important class of high-availability server software running on machines with a small number of cores (up to 16). However, the scalability of the Erlang/OTP system on real applications running on massively multicore machines remains to be carefully studied and proven.

One of the goals of the RELEASE project [8][1] is to improve some core aspects of Erlang's VM in order to make it possible for applications to achieve highly scalable performance on high-end multicore machines of the present and future, with minimal refactoring. A big part of the responsibility for achieving scalability must be lifted from the application programmer and pushed to the VM. Several key components of the VM may have to be redesigned and reimplemented, in case it is found that they currently hinder the scalability of applications on large multicore machines.

To pursue this goal, it is necessary to study the performance and scalability characteristics of a representative set of existing applications running on the current Erlang VM, in order to identify bottlenecks and prioritize changes and extensions in the architectural design of the runtime system and in the reimplementation of key components of the VM. A *benchmark suite* is required for this purpose, whose primary focus will be on being able to effectively measure *scalability*.

This paper presents a scalability benchmark suite for Erlang/OTP. It begins by presenting its similarities and differences with other existing benchmark suites (Section 2) and by setting the scalability dimensions that will be the primary issues under investigation (Section 3). The benchmarking infrastructure is then presented (Section 4), followed by an overview of the benchmarks that have been added to the suite until now (Section 5). In Section 6, we give a summary of the results that we have obtained so far and some of the lessons that we have learnt. Section 7 describes the steps that are required for extending the benchmark suite with new benchmarks, and Section 8 presents some concluding remarks and directions for future work.

Erlang'12, September 14, 2012, Copenhagen, Denmark.
Copyright © 2012 ACM 978-1-4503-1575-3/12/09…$10.00

[1] See also the project's site: `http://www.release-project.eu/`.

2. Related Work

The idea of a benchmark suite that will serve both as a tool to run and analyze benchmarks and as a repository of programs that can be used as benchmarks is not new. In the following paragraphs we analyze a few benchmark suites that have been used extensively or have similar goals with the benchmark suite that we present in this paper.

Basho Bench [3] is the only benchmarking tool that is currently available for the Erlang/OTP. It was originally implemented for Riak [17], but it can be used to benchmark other applications as well. Each benchmark defines one or more types of operations, and the tool is responsible for creating a number of workers, and assign the execution of a number of operations to them. Basho Bench is both configurable and extendable, and produces graphs (throughput and latency), very much like our own suite, but its primary goal is to execute a benchmark in a single rather than in multiple execution environments.

The nofib suite [14] started in the early 1990s as a collection of Haskell programs for benchmarking the implementation of the Glasgow Haskell Compiler. It has since evolved as a benchmark suite geared towards functional languages, oriented mostly towards improving implementations and providing performance comparisons. Due to the variety of benchmarks included, another goal of nofib has been to allow users of the language and a specific implementation to predict the performance of their own programs. A drawback of this benchmark suite is that it does not include tools to analyze the reported performance metrics. Furthermore, the benchmarks included are rather old and it is not clear whether they accurately represent current workloads. The nofib suite has stirred a lot of discussions in the Haskell community, especially with the development of (lots of dialects of) Concurrent and Parallel Haskell. As a result, several other benchmarking suites and benchmarking tools for Haskell have evolved since nofib, such as nobench [12], fibon [15], Criterion [13], and HaBench [9] which is work under progress.

The DaCapo benchmark suite [5] is an enhanceable set of open-source, real and non-trivial Java benchmarks. Although DaCapo focuses on performance rather than scalability, its developers recommend some rather interesting methodologies for the selection, measurement, and evaluation of benchmarks for the Java Virtual Machine. Although these ideas cannot be directly applied to benchmarks for the Erlang Virtual Machine, they have influenced to some extent the way we collect and assess benchmarks for our benchmark suite.

The NAS suite is a suite of parallel performance benchmarks. They were originally developed at the NASA Ames Research Center in 1991 [2] to assess high-end parallel supercomputers. The original NAS Parallel Benchmarks consisted of eight individual benchmark problems, each of which focused on some aspect of scientific computing. In their current incarnation, these benchmarks are implemented using two widely used parallel programming models: MPI and OpenMP. Recently, the benchmarks have been updated with new Grid and "multi-zone" versions, which better reflect modern Computational Fluid Dynamics computations [11]. A significant drawback of the suite is the fact that the implementations of the benchmarks are reference applications and not fully optimized. Furthermore, there is no support in the suite to easily execute sets of experiments and collect and analyze the produced results.

Finally, PARSEC [4] is a recent benchmark suite created to drive the design of the new generation of multiprocessors and multicore systems. Therefore, its main target is to provide applications that have been parallelized with a variety of programming models and are diverse in their characteristics. Furthermore, the benchmarks included in the suite represent emerging workloads that implement state-of-the-art algorithms and are more relevant for contemporary systems. With respect to its overall architecture, PARSEC is similar to our benchmark suite in the sense that it is largely automated, allowing users to create scripts that will run the benchmarks with the requested combinations of input parameters. An additional feature is the inclusion of an instrumentation API that the user can exploit in order to expand the characteristics of a benchmark that are measured.

3. Dimensions in Benchmarking

Quoting from wikipedia:

> *Scalability* is the ability of a system, network, or process, to handle growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth [6]. For example, it can refer to the capability of a system to increase total throughput under an increased load when resources (typically hardware) are added.

It is precisely the last sentence that captures the notion of scalability that we want a user to be able to analyze and measure with our benchmarking suite.

The behaviour of a benchmark is often affected when one or more dimensions of the execution environment change. We begin by identifying the dimensions that are important for scalability benchmarking in massively concurrent Erlang applications. These dimensions define a multi-dimensional space, each point of which corresponds to a different possible configuration of the execution environment.

Number of nodes. This is the number of virtual machines on which the application is executed. Small scale applications typically run on a single node, located on a single physical computer. When computation is distributed in multiple physical machines, e.g., in a cluster or a cloud of computing devices, there will be more than one node. It is also possible to start multiple nodes even on a single physical computer, either using virtualization or even without such separation. Such a configuration usually provides redundancy in cases of failure of a whole node.

Number of cores. A single node running on a multiprocessor computer is capable of parallelizing computation, executing different processes in parallel on different processors or processor cores. Typically, when a node is started, it is configured to use as many CPU cores as are available, but it is possible to restrict the number of CPU cores to a smaller value.

Number of schedulers. A single node may start any number of scheduler OS processes, which in turn handle the execution of Erlang processes. Typically, one scheduler is started for every CPU core (or for every logical core, in case hyper-threading is involved) that is available to the node, with the intention that all schedulers execute in parallel and independently of each other (of course the Erlang node is usually not the only executing application in the OS and therefore this does not always work exactly so). If fewer schedulers are started, then fewer CPU cores will be used. It is also possible, to start more schedulers than the number of available CPU cores; in this case, some cores will be shared by multiple schedulers.

Erlang/OTP release and flavor. Each new Erlang/OTP release contains bug fixes and introduces new features that have a positive (or occasionally a negative) effect on the performance of some applications. It is often the case that developers of the Erlang/OTP want to experiment with different flavors of the VM, different configuration options, different algorithms for implementing the same functionality, and measure the effects of all these.

Command-line arguments to `erl`. The performance measurements obtained for a program may vary, depending on the options that are used to control aspects of the runtime system. An example of such an option is the `+sbt` emulator flag that is used to determine whether and how the schedulers are bound to logical processors.

4. Benchmarking Infrastructure

The benchmark suite is built around two basic notions that need to be explained, not because they are complicated but because they may be used with different meanings in other benchmarking suites or in the literature.

- An *application* is the piece of software whose execution behaviour we intend to analyze and measure. For example, we have used Dialyzer as an application, a piece of software that we further describe in Section 5. Typically, an application consists of a set of Erlang modules, each of which exports a number of functions. There may be a starting point, the application's "main" function, or there may be no such thing and the application will then be considered as a library of modules and functions. It may also be the case that the application we want to study is the Erlang/OTP distribution itself.
- A *benchmark* is a specific use case of the application and includes setting up the environment, calling specific functions in specific ways and using appropriate data for this. There may be more than one benchmarks for a given application, each studying a different use case, e.g., different sizes of input, different functionality of the application, etc. Each benchmark can be executed in multiple points of the multi-dimensional space that we defined in Section 3. Furthermore, a benchmark can be parameterized with a number of *size variables*, which are specific for this benchmark and may be thought of as additional dimensions of the multi-dimensional space.

Our suite thus comprises (i) a set of target applications that we intend to study, (ii) a collection of benchmarks for studying these applications, corresponding to both synthetic and real-world use cases, and (iii) an infrastructure, whose purpose is to build and run the benchmarks and applications, as well as to analyze and compare their performance in various points of the multi-dimensional space to assess scalability. The rest of this section describes the infrastructure, while the next one is dedicated to the benchmarks and their target applications.

4.1 Overview

The benchmarking infrastructure is intended to be a tool that runs benchmarks and collects scalability measurements during their execution. A benchmark can be executed in different *runtime environments* which, as far as the benchmark suite is concerned, are identified by the following parameters:

- the number of Erlang nodes;
- the number of scheduler OS processes on each node;
- the `erl` program used to start each node; and
- the command-line arguments passed to this `erl` program.

In the default runtime environment, there is a single Erlang node with as many schedulers as the number of CPU cores in the system. The node is started using the `erl` program found in the OS path and called with no extra arguments.

If we run a specific benchmark in a specific runtime environment *multiple times*, we may get different measurements each time. Hopefully, the differences will be small, unless the benchmark has a large degree of randomness. In order to eliminate or at least reduce such "noise" in our measurements, we can choose to run a benchmark multiple times and keep statistics, e.g., use the average, the median, the last, or the best value of the elapsed time.

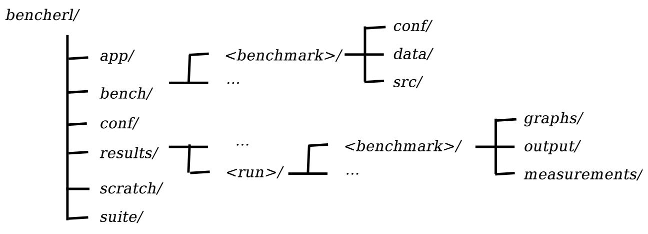

Figure 1. The current directory structure of bencherl.

The benchmark suite can optionally perform a sanity check to verify that the given benchmark has run to completion correctly, in all the different runtime environments that were tested. In order to do that, the suite compares the output that the benchmark produces during its execution in all runtime environments, and the results should normally match.

The scalability measurements that the suite collects during the execution of a benchmark are dumped in files that we can easily read and examine. The suite can also produce scalability graphs that visualize the same information.

For making it possible to alter benchmark sizes easily, each benchmark can have three different versions: a *short*, an *intermediate*, and a *long* one. The different versions of a benchmark differ only in the size parameters, which may depend on the number of available cores and should affect execution time.

Finally, the users of the benchmark suite can specify which benchmark or benchmarks they want to run, in which runtime environments, with which size parameters, etc.

4.2 Current directory structure

This section describes the current directory structure of the benchmark suite, which is shown in Figure 1.

- The `app/` directory is where the source code of the applications resides. Each sub-directory under it corresponds to a different application.
- Inside the `bench/` directory, there is one sub-directory for each benchmark in the suite. In general, a benchmark corresponds to an application in `app/` but we have used two different trees to make it simpler to include synthetic benchmarks that are self-contained, or even to include benchmarks that test larger pieces of existing software, possibly not written in Erlang, that are not part of the suite. We will give more details about the internal structure of a benchmark directory in Section 7, where we will talk about how the suite can be enhanced with new benchmarks.
- The `conf/` directory contains, among other things, a file named `run.conf`. This is the run configuration file that contains all the information that the suite needs, in order to know what to execute and how.
- For each run of the benchmark suite, a new sub-directory is created in the `results/` directory. The name of this sub-directory can either be a mnemonic name that we choose for this run, or just the date and time when the run started. In there, one sub-directory is created for each benchmark that was executed, which in turn contains the following sub-directories:
  - `graphs/`: it contains the scalability graphs that the suite produced;
  - `output/`: it contains the output that the benchmark produced during its execution; and

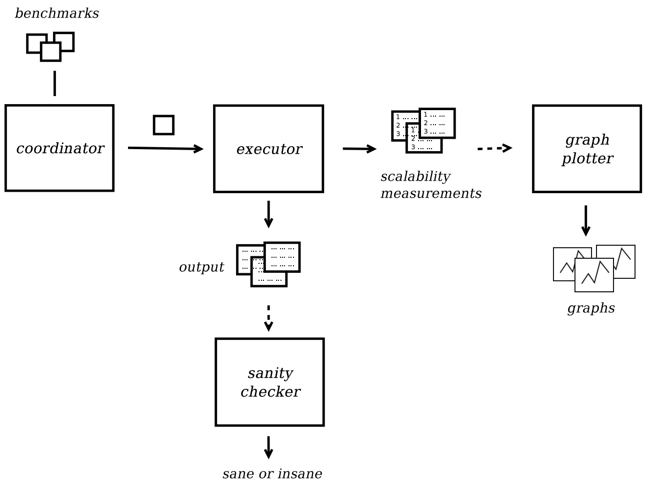

Figure 2. The architecture of bencherl.

  - `measurements/`: it contains the scalability measurements that were collected during the execution of the benchmark.

- The `scratch/` directory is used by the suite as a place to store temporary files.
- Finally, the `suite/` directory contains the code that implements the benchmarking infrastructure.

4.3 Architecture and what goes on behind the scenes

The benchmark suite consists of four components: the *coordinator*, the *executor*, the *sanity checker*, and the *graph plotter*. Figure 2 shows how these components interact with each other. Each one of these components is described in detail in the rest of this section.

4.3.1 The coordinator

In each run of the suite, the coordinator reads the configuration file (i.e., `conf/run.conf`) in order to find out what to execute and how (e.g., which benchmarks, what version of the benchmarks in case there are multiple versions, in which execution environment, etc.). It creates a new sub-directory in `results/` for the results of this run and starts the execution of the benchmarks.

Before executing a benchmark, the coordinator reads the benchmark's configuration file for changes to the execution plan for this specific benchmark (the settings specified in a benchmark's configuration file always override those specified in the global configuration file of the suite).

For each runtime environment in which the benchmark must run, the coordinator sets up the environment, performs any benchmark-specific pre-execution actions, and invokes the *executor* to handle the actual execution of the benchmark. When the executor has finished its job, the coordinator takes over again, performs any benchmark-specific post-execution actions, and might repeat the execution of the benchmark in some other runtime environment.

Once the execution of the benchmark has completed, the coordinator asks the *sanity checker* to verify the correctness of the output that the benchmark produced during its execution. It also asks the *graph plotter* to use the collected measurements to produce scalability graphs, and then continues with the next benchmark.

4.3.2 The executor

The executor is responsible for the execution of a specific benchmark in a specific runtime environment. The coordinator has prepared a configuration file with instructions on what the executor must do, and has put it in the suite's `scratch/` directory for the executor to find and use.

The runtime environment has already been partially set up by the coordinator: an Erlang runtime system has been started with the appropriate number of schedulers, using the appropriate version of the `erl` program, and passing the appropriate command line arguments to it. The runtime environment, in which the executor must run the benchmark, is the environment in which the executor itself runs. So, all that is left for the executor to do is start the appropriate slave nodes, if any.

Now that the runtime system is ready, the executor runs the appropriate version (short, intermediate, long) of the benchmark. Execution of the benchmark's code takes place in a new process, which notifies the executor as soon as it is complete. Thus, the executor knows the running time of the benchmark. The executor repeats the benchmark execution as many times as specified, and keeps statistics. In order to measure time, the executor uses the `erlang:now/0` and `timer:diff/2` functions.

Finally, the executor makes sure that the output that the benchmark produces during its execution is stored in the appropriate file in the `output/` sub-directory of the benchmark's result directory (so that the sanity checker can later find and use it), and that the collected measurements end up in the appropriate file in the `measurements/` sub-directory of the benchmark's result directory (so that it can later be processed by the graph plotter).

4.3.3 The sanity checker

When a benchmark runs, it may produce output. In case the benchmark is deterministic and the output depends only on the given input, the output should be the same across all the executions of the benchmark in all the runtime environments. The sanity checker, which is a component of the benchmark suite, can verify that.

The suite considers everything that the benchmark writes on the standard output during its execution as the benchmark's output. All this information is stored in a file in the `output/` sub-directory of the benchmark's result directory. There is one file in this directory for each runtime environment, in which the benchmark runs.

What the sanity checker does is compare all the files that reside in the benchmark's `output/` directory.[2] If there are any differences between any pair of output files, then the sanity check is considered to have failed.

4.3.4 The graph plotter

The graph plotter processes the scalability measurements that have been collected during the execution of a benchmark. It produces graphs that visualize all this information in a way that can help us get a feeling of how well a benchmark scales.

The scalability measurements gathered from the execution of a benchmark are stored in files in the `measurements/` sub-directory of the benchmark's result directory. In particular, this directory contains two files that correspond to the execution of the benchmark with a specific `erl` program and with specific command-line arguments: one with scalability measurements for the execution of the benchmark with different number of schedulers, and one with scalability measurements for the execution of the benchmark with different number of nodes.

The graph plotter produces one graph for each one of the files in the benchmark's `measurements/` directory. Each one of these graphs shows how the increase of schedulers (or the increase of

[2] For the time being, the `diff` application is used for comparing files, and differences in white space are ignored. If needed, it should be easy however, to extend the sanity checker with custom ways of validating the sanity of output files, e.g., by running a benchmark-specific piece of code.

nodes) affects the behaviour of the benchmark, when the benchmark is executed with the same `erl` program and with the same command-line arguments.

In case the benchmark was executed using more than one `erl` program (i.e., with different releases of Erlang/OTP), the graph plotter combines the files that concern a specific set of command-line arguments and produces a graph that compares the behaviour of the benchmark when executed with different `erl` programs, but with the same command-line arguments.

The same happens if the benchmark was executed using more than one command-line argument sets. The graph plotter uses the scalability measurements collected during the execution of the benchmark with a specific `erl` program, and produces a graph that shows how the different command-line arguments affect the scalability of the benchmark, when all the other parameters of the runtime environment remain the same.

Finally, the graph plotter produces speed-up graphs, apart from time graphs, to make the results easier to understand from the scalability point of view.

5. Benchmarks

The benchmark suite comes with an initial collection of parallel and distributed benchmarks. We classify these benchmarks into two categories:

- *Synthetic benchmarks*, which are typically small pieces of code that measure the execution behaviour of some specific aspect of Erlang execution (e.g., spawning processes, message passing, etc.). Most of these benchmarks do not correspond to applications but are self-contained.
- *Real-world benchmarks*, which typically study various aspects of the behaviour of large, existing Erlang applications.

In the rest of this section, we give a brief description of each one of the suite's benchmarks.

5.1 Synthetic benchmarks

The benchmarks in this category can be used to measure scalability of specific aspects of Erlang-style concurrency and expose possible bottlenecks in the virtual machine of Erlang/OTP, e.g., the (ab)use of the `erlang:now/0` function — see Section 6.

bang: A benchmark for many-to-one message passing that spawns one receiver and multiple senders that flood the receiver with messages. The parameters of the benchmark are the number of senders to spawn and the number of messages that each sender will send to the receiver.

big: A benchmark that implements a many-to-many message passing scenario. Several processes are spawned, each of which sends a `ping` message to the others, and responds with a `pong` message to any `ping` message it receives. The benchmark is parameterized by the number of processes.

ehb: This is an implementation of hackbench [20] in Erlang, a benchmark and stress test for Linux schedulers. The number of groups and the number of messages that each sender should send to each receiver in the same group are the two parameters that this benchmark receives.

ets_test: This benchmark creates an ETS table and spawns several readers and writers that perform a certain number of reads (lookups) and writes (inserts), respectively, to that table. The benchmark is parameterized by the number of readers, the number of writers and the number of operations (insert/lookup) that each reader or writer should perform.

genstress: This is a generic server (`gen_server`) benchmark that spawns an echo server and a number of clients. Each client fills its message queue with a number of dummy messages; it then sends some messages to the echo server and waits to get them back as a response from the server. The benchmark can be executed with or without using the `gen_server` behaviour, as well as with a different number of clients, dummy messages and messages exchanged with the echo server.

mbrot: This benchmark extrapolates the coordinates of a 2-D complex plane that correspond to the pixels of a 2-D image of a specific resolution. For each one of these points, the benchmark determines whether the point belongs to the Mandelbrot set or not. The total set of points is divided among a number of workers. The benchmark is parameterized by the dimensions of the image.

orbit_int: The *orbit problem* is defined as follows: Given a space X, a list of generators $f_1, ..., f_n : X \to X$ and an initial vertex $x_0 \in X$, compute the least subset $Orb \subseteq X$, such that $x_0 \in Orb$ and Orb is closed under all generators. We consider a special case of the orbit problem, where X is a finite subset of the natural numbers. This benchmark operates on a distributed hash table, and follows a master/worker architecture. The master initiates the computation, and waits for its termination. Each worker hosts a chunk of the hash table. When a worker receives a vertex, it stores it into its chunk, applies all generators to it, and then sends the generated vertices to the corresponding nodes that are responsible for them. The master and all the workers are processes on the same or on different Erlang nodes. The benchmark creates the hash table, distributes it evenly across the workers, and computes the orbit in parallel. The parameters of the benchmark are a list of generators, the size of the space, the number of workers, a list of nodes to spawn workers on, and whether there will be intra-worker parallelism or not.

parallel: A benchmark for parallel execution that spawns a number of processes, each of which creates a list of N timestamps and, after it checks that each element of the list is strictly greater than its previous one (as promised by the implementation of `erlang:now/0`), it sends the result to its parent. The benchmark is parameterized by the number of processes and the number of timestamps.

pcmark: This benchmark is also about ETS operations. It creates five ETS tables, fills them with values, and then spawns a certain number of processes that read the contents of those tables and update them. As soon as one process finishes, a new process is spawned, until a certain total number of processes has been reached. The benchmark is parameterized by the number of initial processes and the total number of processes.

ran: Another benchmark for parallel execution that spawns a certain number of processes, each of which generates a list of 10000 random integers, sorts it and sends its first half to the parent process. The benchmark receives the number of processes as a parameter.

serialmsg: A benchmark about message proxying through a dispatcher. The benchmark spawns a certain number of receivers, one dispatcher, and a certain number of generators. The dispatcher forwards the messages that it receives from generators to the appropriate receiver. Each generator sends a number of messages to a specific receiver. The parameters of the benchmark are the number of receivers, the number of messages and their length.

timer_wheel: A timer management benchmark that spawns a certain number of processes that exchange `ping` and `pong` messages. Each process sends a `ping` message to all other processes, and then waits (with or without a timeout) to receive a `pong` message as a response. In the meantime, the process responds with a `pong` message to any `ping` message it receives. In case of a timeout, the corresponding process dies. The benchmark is parameterized by the number of processes.

5.2 Dialyzer benchmarks

Dialyzer [10] is a static analysis tool that identifies software discrepancies (e.g., definite type errors, unreachable code, redundant tests) in single Erlang modules or entire applications. It is designed to be sound for defect detection, meaning that all the warnings it emits regard errors that the developer can (and should) act upon (i.e., it produces no false positives). This has made Dialyzer one of the most widely used tools among Erlang developers.

Dialyzer assigns a type to each function in each module that is given as input, using a combination of constraint-based type inference and dataflow analysis. A bottom-up approach is used, starting with functions that have no calls and moving higher in the call graph until all the functions have been processed. This was originally done sequentially, with the functions being processed in reverse topological order[3]. After types have been inferred, they are used to detect discrepancies in the code base which is analyzed.

A parallel version of Dialyzer was developed [1], exploiting the possibility to concurrently infer types for functions that only have calls to functions that have already been analyzed. A separate Erlang process is spawned for each function and then message passing is used to determine when the dependencies of each function have been processed. The types inferred are stored in an ETS table. Parallel Dialyzer is therefore an application that has a large number of processes, communicating freely with small messages and relying on ETS to store data shared among processes.

dialyzer_bench: This benchmark is divided in two separate parts, executed one after the other. These two parts correspond to the two most common use cases of the Dialyzer tool.

- The first part measures the time required for the generation of a Persistent Lookup Table (PLT) that includes the main applications in the Erlang/OTP distribution. This PLT contains the standard set of “trusted code” that most Erlang developers rely upon when analyzing their own applications.
- The second part measures the time required for the actual analysis of most of applications in the Erlang/OTP distribution.

5.3 Scalaris benchmarks

Scalaris [16, 19] is an Erlang implementation of a distributed key-value store, which has been designed for good horizontal scalability, i.e., good performance for simple read/write operations distributed over many servers. It is based on the Chord# protocol [18], and supports fail-over, data and service distribution and replication, strong consistency and transactions.

The nodes are connected in a ring topology and store their data in memory. When data have to be distributed over nodes, key ranges are assigned to nodes, instead of the more typical approach that uses hashing to distribute the data. As a result, a query does not need to go to every node and if the distribution of keys is performed carefully it may allow for better load balancing. Replication of data on several nodes also helps towards this goal. The combination of the ring topology and replication allows every read/write of a key/value pair to complete in $\log(N)$ time, where N is the number of nodes.

scalaris_bench: Scalaris has been successfully used to implement a number of applications, like Web 2.0 services, in a distributed manner [16]. A common operation in these cases is to read data that are requested from clients. Thus, this benchmark creates a ring with a certain number of Scalaris nodes, and spawns a certain number of processes on each one of them. Each process picks a random key and reads its value a certain number of times. The benchmark has three parameters: the number of Scalaris nodes, the number of processes to spawn on each node, and the number of reads that each process executes.

5.4 Sim-Diasca benchmark

Sim-Diasca [7] stands for Simulation of Discrete Systems of All Scales. It is a lightweight simulation platform, released by EDF R&D under the GNU LGPL, offering a set of simulation elements, including notably a simulation engine, to be applied to the simulation of discrete event-based systems made of potentially numerous interacting parts. This class of simulation encompasses a wide range of target systems, from ecosystems to information systems, i.e., it could be used for most applications in the so-called *complex systems* scientific field. A classical use case for Sim-Diasca is the simulation of industrial distributed systems federating a large number of networked devices.

The simulator is designed to be parallel both on a single node (multiple models can be evaluated simultaneously by the available cores) and on a distributed system (a simulation can take place over a set of networked computation nodes), and all this without hurting the properties deemed important but difficult to preserve in that context, such as:

- the correctness of the evaluation of models;
- the preservation of causality between simulation events; and
- the ability to have completely reproducible simulations.

Sim-Diasca can be used to simulate any kind of complex discrete system and has so far been used successfully to model two projects related to the massive roll-out of communicating meters for millions of residential customers. This flexibility comes from the fact that the simulation engine is completely separate from the models it can support and Sim-Diasca offers an extensive library for the development of models.

sim_diasca_bench: The benchmark of Sim-Diasca is based on a real-world scenario of power distribution networks, but it is still under development as sensitive information needs to be cleaned out before it can be included in our public suite. The simulator has already been tested with a mock model of the network but this model is too lightweight to provide any useful scalability measurements.

6. Scalability Results

We now describe some of the experiments that we performed using the benchmark suite. In this context, the experiments themselves are not important, except as use cases for the benchmark suite. All benchmarks were run on a machine with four AMD Opteron 6276 (2.3 GHz, 16 cores, 16M L2/16M L3 Cache), giving a total of 64 cores and 128 GB or RAM, running Linux 3.2.0-amd64.

Experience #1: Some benchmarks already scale well.

A typical representative of our first experience is the big benchmark. We executed the long version of the benchmark (1536 processes) using 1 to 64 schedulers in Erlang/OTP R15B01, and we

[3] As functions might form call cycles, this reverse topological ordering is actually obtained from the condensation of the call graph into strongly connected components.

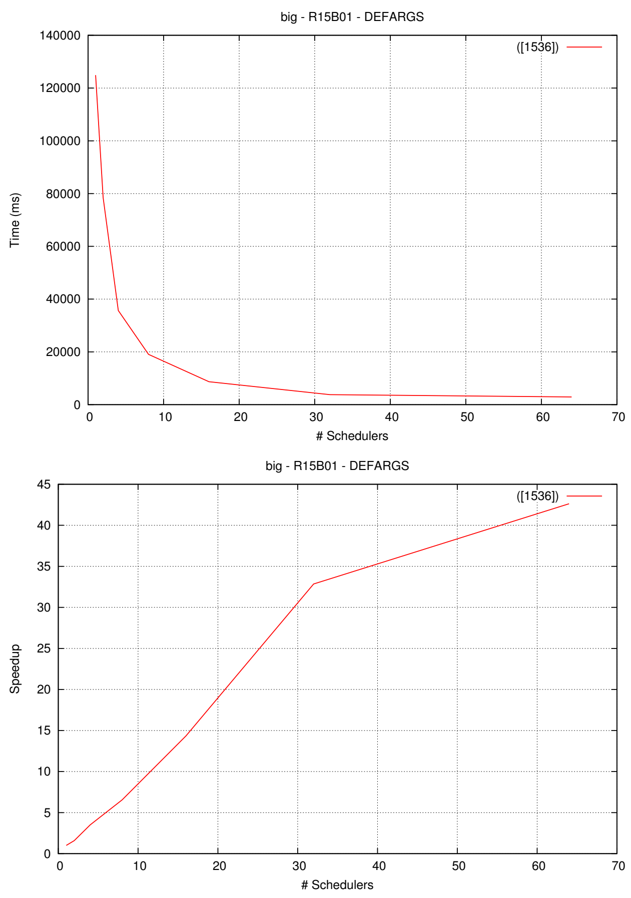

Figure 3. Running the long version of the big benchmark with Erlang/OTP R15B01 on a different number of schedulers.

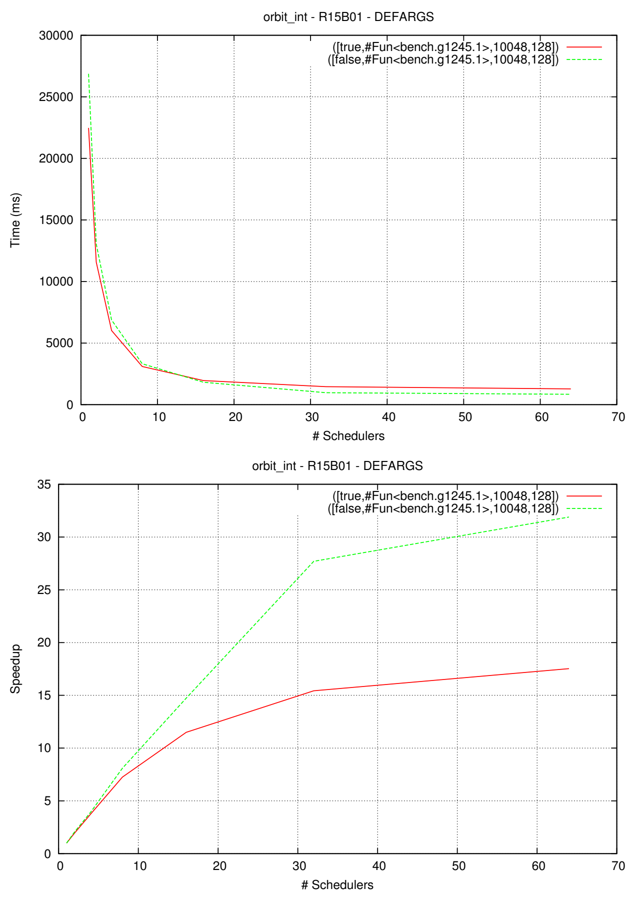

Figure 4. Running the long version of the orbit_int benchmark with Erlang/OTP R15B01 on a different number of schedulers, with and without intra-worker parallelism.

specified a number of 5 iterations, in order to get more accurate measurements. The results are shown in Figure 3 and it is obvious why big is the most used benchmark in presentations that show how well Erlang scales on big multicores. Its speedup reaches a factor of 43 with 64 schedulers.

Experience #2: Some benchmarks scale well only in one node.

We experimented with the orbit_int benchmark on two dimensions: (a) on a single node with 1 to 64 schedulers, and (b) on 1 to 64 nodes with 1 scheduler each. We used the long version of the benchmark (space of size 10048, 128 workers, 4 generators, with or without intra-worker parallelism) in both experiments. Based on the results of the first experiment, shown in Figure 4, the execution time of the benchmark continuously decreases as we add more schedulers to it. On the other hand, the behaviour of the benchmark, when we used more than one nodes to execute it, was not what we expected (see Figure 5); the execution time increases when 8 or more nodes are used and we suspect that creating more nodes on a single machine is not as effective as using more schedulers..

Experience #3: Some benchmarks do not scale.

A different set of benchmarks shows that some concurrent programs do not scale well at all, some for reasons that are obvious and some for reasons that may elude us for long. The parallel benchmark belongs to the latter subcategory. As Figure 6 shows, with each additional core, we witness a significant slowdown in this benchmark, up to 8 cores where something changes and we have a small speedup until 32 cores, and then again a slowdown.

The benchmark is very simple and seems to be very similar to ran, which scales almost as well as big. There is a very small difference, easily overlooked, that explains this behaviour. In order to make timestamps, parallel invokes the function `erlang:now/0`, whose implementation acquires a global lock for returning a unique timestamp. This lock is precisely the bottleneck in the VM that obstructs the scalability for this benchmark.

Experience #4: Different command-line options and scalability.

Dialyzer is an example of an application that scales relatively well. In order to investigate how the scheduler bind type affects scalability, we ran the dialyzer_bench benchmark with two different values for the `+sbt` emulator flag which is used to determine the scheduler bind type: with the value `u`, which indicates that schedulers will not be bound to logical processors, and with the value `tnnps`, which indicates that schedulers will be spread over hardware threads across NUMA nodes, but only over processors of one NUMA node at a time. As shown in Figure 7, the dialyzer_bench benchmark performs better when there is no binding between schedulers and logical processors.

Experience #5: Different Erlang/OTP releases and scalability.

In April 2012, there were complaints on the Erlang mailing list from the development team of Scalaris that the performance of their application in Erlang/OTP R15B and R15B01 dropped, in comparison to its performance in R14B04. In order to reproduce the experience reported, we ran the long version of the scalaris_bench

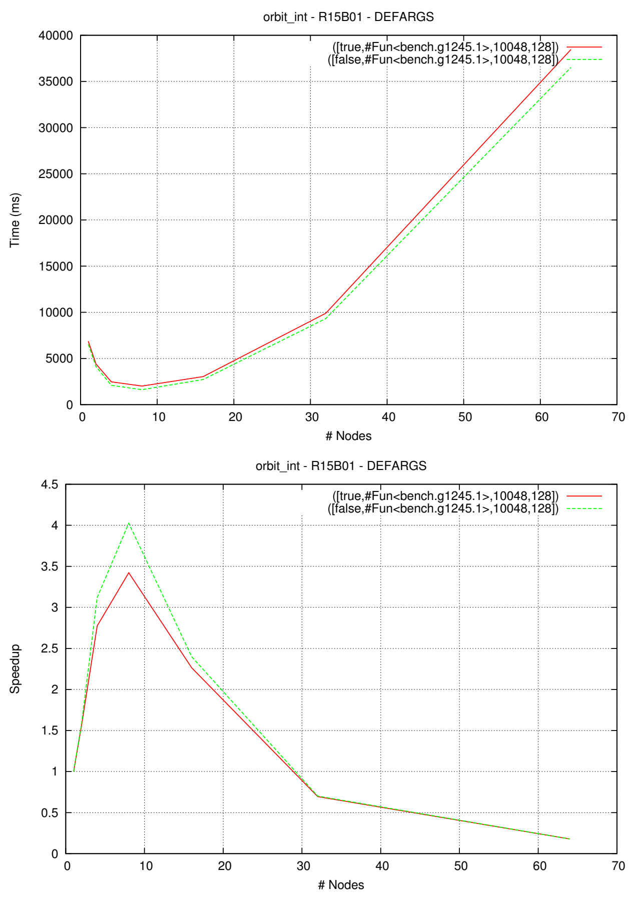

Figure 5. Running the long version of the orbit_int benchmark with Erlang/OTP R15B01 on a different number of nodes with 1 scheduler each, with and without intra-worker parallelism.

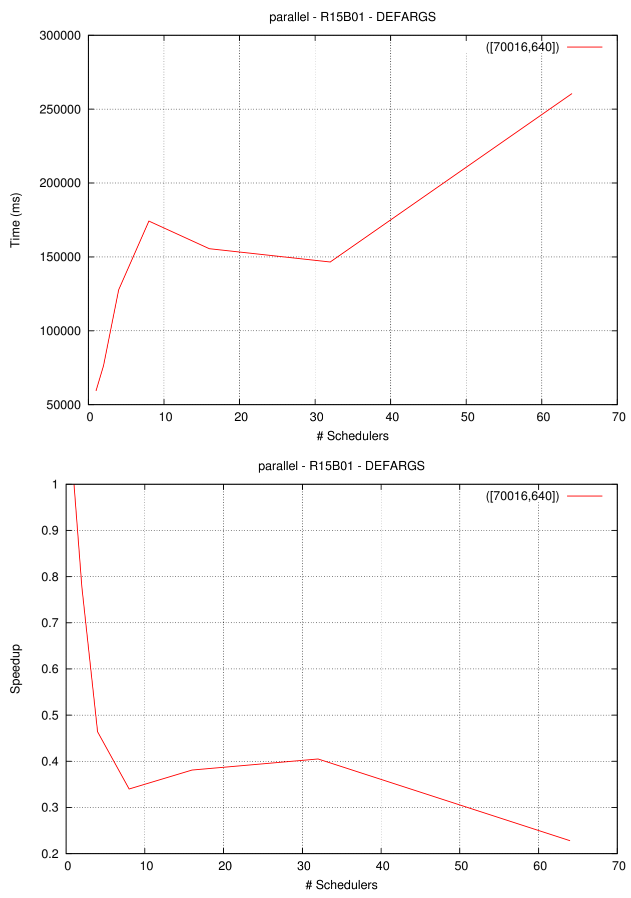

Figure 6. Running the long version of the parallel benchmark with Erlang/OTP R15B01.

benchmark (32 nodes, 64 threads per node, 1024 reads per thread). Figure 8 shows that the performance of Scalaris indeed deteriorates in Erlang/OTP R15B and R15B01, compared to its performance in R14B04, especially beyond 16 schedulers.

7. Extending the Benchmark Suite

As mentioned, the benchmark suite is designed to be enhancable with new benchmarks, both synthetic and real-world. This section describes the steps that must be followed to add a new benchmark to the suite.

If the benchmark is written for a real-world, open-source application, one can add this application in a directory under the suite's `app/` directory. This helps the benchmark suite to be self-contained. It also allows the target application to be built and run with the same Erlang/OTP as the benchmark.

The first step is to create a directory for the benchmark under `bench/`. This is where everything related to the new benchmark will end up.

The next step is to create a `src/` sub-directory in the new benchmark directory and put there the benchmark's source code. One needs to write a handler for the benchmark, which needs to be added to the same directory. A *benchmark handler* is a standard Erlang module that has the same name as the benchmark and exports two functions: `bench_args/2` and `run/3`. It is essentially the module with which the executor interacts when it executes this benchmark.

Function `bench_args/2` has the following signature:

```
bench_args(Vrsn, Conf) -> Args
  when
    Vrsn :: 'short' | 'intermediate' | 'long',
    Conf :: [{Key :: atom(), Val :: term()}, ...],
    Args :: [[term()]].
```

It returns the different argument sets that should be used to run this version (`Vrsn`) of the benchmark. Information from the configuration (`Conf`) of the benchmark can be used (e.g., the number of available cores), in order to generate an appropriate argument set for each execution environment. For example, the orbit_int benchmark defines `bench_args/2` as shown below:

```
bench_args(Vrsn, Conf) ->
  {_,Cores} = lists:keyfind(number_of_cores, 1, Conf),
  [G0,N0,W0] = bench_args_aux(Vrsn),
  [[IWP,G,N,W] || IWP <- [true,false], G <- [G0],
                  N <- [N0 * Cores], W <- [W0 * Cores]].

bench_args_aux(Vrsn) ->
  case Vrsn of
    short -> [fun bench:g13/1, 11, 2];
    intermediate -> [fun bench:g124/1, 157, 2];
    long -> [fun bench:g1245/1, 157, 2]
  end.
```

The parameters `G`, `N` and `W` are different for each size. On the other hand, the benchmark is run both with enabled and with disabled

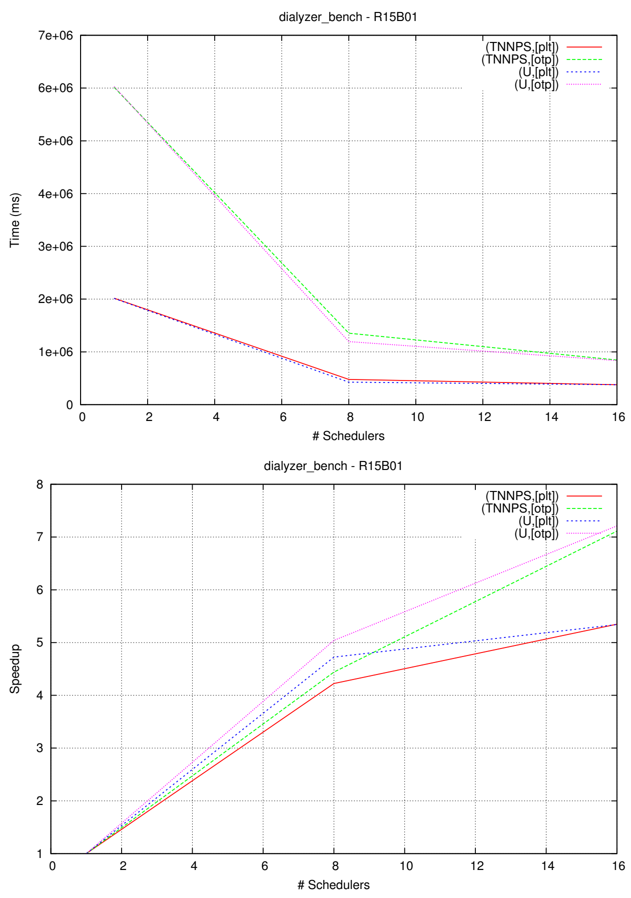

Figure 7. Running the dialyzer_bench benchmark, first to build a PLT and then to analyze all the Erlang libraries, with Erlang/OTP R15B01 and different scheduler bind types (`u` and `tnnps`).

intra-worker parallelism (i.e., with the parameter `IWP` being first set to `true` and then to `false`).

Function `run/3` has the following signature:

```
run(Args, Slaves, Conf) -> 'ok' | {'error', Reason}
  when
    Args   :: [term()],
    Slaves :: [node()],
    Conf   :: [{Key :: atom(), Val :: term()}, ...],
    Reason :: term().
```

It uses the arguments in `Args`, the slave nodes in `Slaves` and the settings in `Conf` to run the benchmark. For example, the following is the definition of `run/3` in the orbit_int benchmark:

```
run([true,G,N,W|_], [], _) ->
  io:format("~p~n", [bench:par(G,N,W)]);
run([false,G,N,W|_], [], _) ->
  io:format("~p~n", [bench:par_seq(G,N,W)]);
run([true,G,N,W|_], Slaves, _) ->
  io:format("~p~n", [bench:dist(G,N,W,Slaves)]);
run([false,G,N,W|_], Slaves, _) ->
  io:format("~p~n", [bench:dist_seq(G,N,W,Slaves)]).
```

This function ignores `Conf`, and uses `Args` and `Slaves`, in order to calculate the orbit and display its size. Depending on whether intra-worker parallelism is enabled or not, and whether the benchmark is to run on multiple nodes, the appropriate function of module `bench` is called.

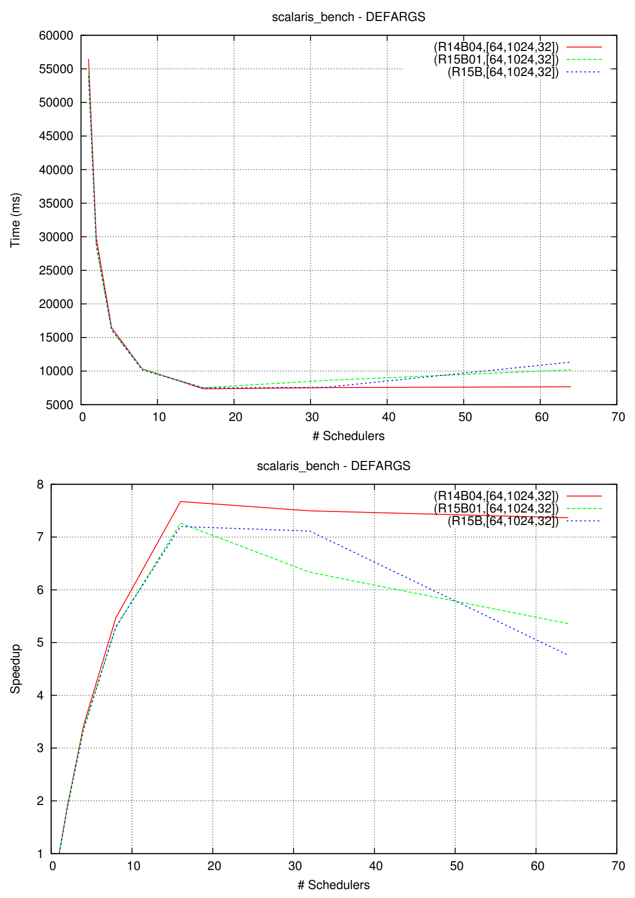

Figure 8. Running the long version of scalaris_bench, with Erlang/OTP R14B04, R15B and R15B01.

Each benchmark directory can also contain a `conf/` directory. If the user wants to specify a specific run configuration for the benchmark (that will override the run configuration of the suite), he can add a `bench.conf` file in the `conf/` sub-directory of the benchmark directory, and specify all the settings in it. Apart from this file, the `conf/` directory can also contain two more files: a `pre_bench` and `post_bench` file. These files serve as “hooks” that are called before and after the execution of a benchmark in a new runtime environment, respectively. Finally, in case the new benchmark needs external data, these can be placed in a `data/` sub-directory.

8. Concluding Remarks and Future Work

We introduced a benchmark suite for Erlang applications that can be used to analyze their scalability, rather than their performance. We have identified nodes, cores, schedulers, Erlang/OTP versions and `erl` command-line arguments (i.e., run options) as the different dimensions that might affect the scalability of an application.

Although at the moment we can use the suite to get a pretty good idea about whether an application scales well under certain conditions, if an application does not scale well, the suite does not provide enough information to understand what is actually wrong. Thus, we are planning to extend the suite so that it collects more information during the execution of a benchmark (perhaps by using the tracing mechanism of Erlang or by taking advantage of the DTrace support that has been recently added in Erlang/OTP).

Acknowledgments

This work has been supported by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software".

We thank Kenneth Lundin, Rickard Green and Patrik Nyblom for sending us the first versions of the synthetic benchmarks, Patrick Maier for sending us the Orbit benchmark, Nico Cruber for helping us with the benchmarking of Scalaris, and Olivier Boudeville for Sim-Diasca.

References

[1] S. Aronis and K. Sagonas. On using Erlang for parallelization: Experience from parallelizing dialyzer. In draft proceedings of the Symposium on Trends in Functional Programming, June 2012.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, R. L. Carter, T. A. Lasinski, D. S. Browning, L. Dagum, R. A. Fatoohi, P. O. Frederickson, and R. S. Schreiber. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[3] Basho Bench. Basho: Benchmarking. URL `http://wiki.basho.com/Benchmarking.html`.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Jan. 2011.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press. doi: 10.1145/1167473.1167488.

[6] A. B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd International Workshop on Software and Performance*, pages 195–203, Ottawa, Ontario, Canada, 2000. ACM. doi: 10.1145/350391.350432.

[7] O. Boudeville. *Technical Manual of the Sim-Diasca Simulation Engine*. EDF R&D, 2012.

[8] O. Boudeville, F. Cesarini, N. Chechina, K. Lundin, N. Papaspyrou, K. Sagonas, S. Thompson, P. Trinder, and U. Wiger. RELEASE: a high-level paradigm for reliable large-scale server software. In draft proceedings of the Symposium on Trends in Functional Programming, June 2012.

[9] HaBench. Habench: Haskell benchmark suite. URL `http://www.haskell.org/haskellwiki/HaBench`.

[10] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In W.-N. Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2004. ISBN 3-540-23724-0. doi: 10.1007/978-3-540-30477-7_7.

[11] NAS. NAS parallel benchmarks. URL `http://www.nas.nasa.gov/publications/npb.html`.

[12] nobench. The nobench suite of benchmark programs. URL `http://code.haskell.org/nobench/`.

[13] B. O'Sullivan. Criterion, an improved Haskell benchmarking library. URL `http://www.serpentine.com/blog/2009/11/06/criterion-0-2-an-improved-haskell-benchmarking-library`.

[14] W. Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag. ISBN 3-540-19820-2.

[15] D. M. Peixotto. The fibon package. URL `http://hackage.haskell.org/package/fibon`.

[16] A. Reinefeld, F. Schintke, T. Schütt, and S. Haridi. A scalable, transactional data store for future internet services. In G. Tselentis, J. Domingue, A. Galis, A. Gavras, D. Hausheer, S. Krco, V. Lotz, and T. Zahariadis, editors, *Towards the Future Internet: A European Research Perspective*, pages 148–159. IOS Press, 2009. ISBN 978-1-60750-007-0. doi: 10.3233/978-1-60750-007-0-148.

[17] Riak. Riak. URL `http://wiki.basho.com/Riak.html`.

[18] T. Schütt, F. Schintke, and A. Reinefeld. A structured overlay for multi-dimensional range queries. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 503–513. Springer, 2007. ISBN 978-3-540-74465-8.

[19] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional P2P key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on Erlang*, pages 41–48, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4. doi: 10.1145/1411273.1411280.

[20] Y. Zhang. Hackbench, 2008. URL `http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c`.