Distributed Computing (2022) 35:439–454
https://doi.org/10.1007/s00446-022-00429-7

PerformERL: a performance testing framework for erlang

Walter Cazzola[1] · **Francesco Cesarini**[2] · **Luca Tansini**[1]

Received: 3 March 2020 / Accepted: 24 May 2022 / Published online: 1 August 2022

Abstract
The Erlang programming language is used to build concurrent, distributed, scalable and resilient systems. Every component of these systems has to be thoroughly tested not only for correctness, but also for performance. Performance analysis tools in the Erlang ecosystem, however, do not provide a sufficient level of automation and insight needed to be integrated in modern tool chains. In this paper, we present `PerformERL`: an extendable performance testing framework that combines the repeatability of load testing tools with the details on how the resources are internally used typical of the performance monitoring tools. These features allow `PerformERL` to be integrated in the early stages of testing pipelines, providing users with a systematic approach to identifying performance issues. This paper introduces the `PerformERL` framework, focusing on its features, design and imposed monitoring overhead measured through both theoretical estimates and trial runs on systems in production. The uniqueness of the features offered by `PerformERL`, together with its usability and contained overhead prove that the framework can be a valuable resource in the development and maintenance of Erlang applications.

Keywords Erlang · Distributed systems · Performance testing · Load testing · Performance monitoring

1 Introduction

Erlang offers a set of features—such as share-nothing lightweight processes and asynchronous communication through message passing—making it the ideal programming language for building massively concurrent systems [10]. Applications running inside the Erlang virtual machine (called the BEAM) use this concurrency model for distribution, resilience and scalability [11]. But with the advent of new technologies—such as cloud computing, containerization and orchestration—developers are not encouraged to be resource savvy in order to satisfy their scalability requirements. This approach implies that performance issues and bottlenecks often go undetected during the development process, only to be identified when the system is in production. As discussed by Jiang and Hassan [21], fixing these issues when in production becomes complicated and expensive.

Several language agnostic tools are available to measure the throughput and latency of a system under test (SUT) by simulating different loads and monitoring response times. These tools—dubbed *load testing tools*—provide system usability metrics and enable the repeatability of the trial runs. But as they use an external observation point (black-box approach), they are not informative on how resources are used inside the SUT. This testing approach can help detecting performance degradation, but provides little information over which component of the SUT is causing the degradation. *Performance monitoring tools*, on the other hand, can gather detailed metrics about the resources used by the SUT, such as memory consumption, CPU usage and I/O operations. Unfortunately, they do not provide an interface to generate load, as they are meant for the inspection of live production systems and are manually added at a later stage of the development. This lack of support for writing automated and repeatable performance tests means that *performance monitoring tools* cannot easily be included as part of the testing pipeline in the development stages.

This paper proposes `PerformERL`, a *performance testing* framework for Erlang that combines the two approaches. The *performance testing* terminology and the distinction be-

✉ Walter Cazzola
cazzola@di.unimi.it

Francesco Cesarini
francesco@erlang-solutions.com

Luca Tansini
luca.tansini@studenti.unimi.it

[1] Department of Computer Science, Università degli Studi di Milano, Milan, Italy

[2] Erlang Solutions, London, United Kingdom

tween *load testing* and *performance monitoring* was first outlined by Gheorghiu [14] and later refined by Jiang and Hassan in their survey [21]. PerformERL enables programmers to write a systematically repeatable suite of tests that stress test the SUT in the early stages of development and keep track of the performance of every component—in terms of resource utilization—as the codebase grows.

PerformERL builds on top of the Erlang BEAM, copes with Erlang ecosystem and exploits the BEAM tracing infrastructure. Its main contribution is to define an architecture and a methodology to enable the performance testing in the Erlang ecosystem. To the best of our knowledge, PerformERL is the first framework in the Erlang ecosystem that permits to programmatically exercise a SUT and gather detailed metrics about the performance of the SUT, how the resources are used by the SUT components and which component and/or resource usage is responsible of the performance degradation of the SUT. Such a contribution is achieved through the design of a specific architecture (details in Sects. 2 and 3) and in some extensions to the tracing infrastructure in order to improve its applicability and performance (details in Sects. 3.4.1 and 3.4.2). Also the proposed architecture is general enough to be implemented in different ecosystems as explained in Sect. 3.6.

The rest of this paper is organized as follows. Section 2 provides an overview of the main concepts and terminology of PerformERL. Section 3 describes the internal architecture of the framework and how it can be realized in the JVM ecosystem. Section 4 shows how PerformERL can be employed and extended with some examples. In Sect. 5, some theoretical measurements and tests to study PerformERL overhead and performance are presented and their results are discussed. Sections 6 and 7 conclude the paper reviewing with related work and presenting our conclusions.

2 Overview

PerformERL is a *performance testing* framework. According to Jiang and Hassan [21], it is neither a load testing nor a performance monitoring tool, but a bit of both. It combines the repeatability of load testing with the visibility offered by a performance monitor. PerformERL should be used as any other testing tool: by writing a test suite dedicated, this time, to performance evaluation. The test files (sometimes also referred to as load generator files) written by the users implement callbacks (defined in the load generator behavior, see Listing 1, details in Sect. 3.1), used by PerformERL to (i) exercise a specific execution of the SUT in which the performance measurements will be gathered, (ii) generate the target function patterns, and (iii) set other configuration parameters, such as size, name and duration of the test.

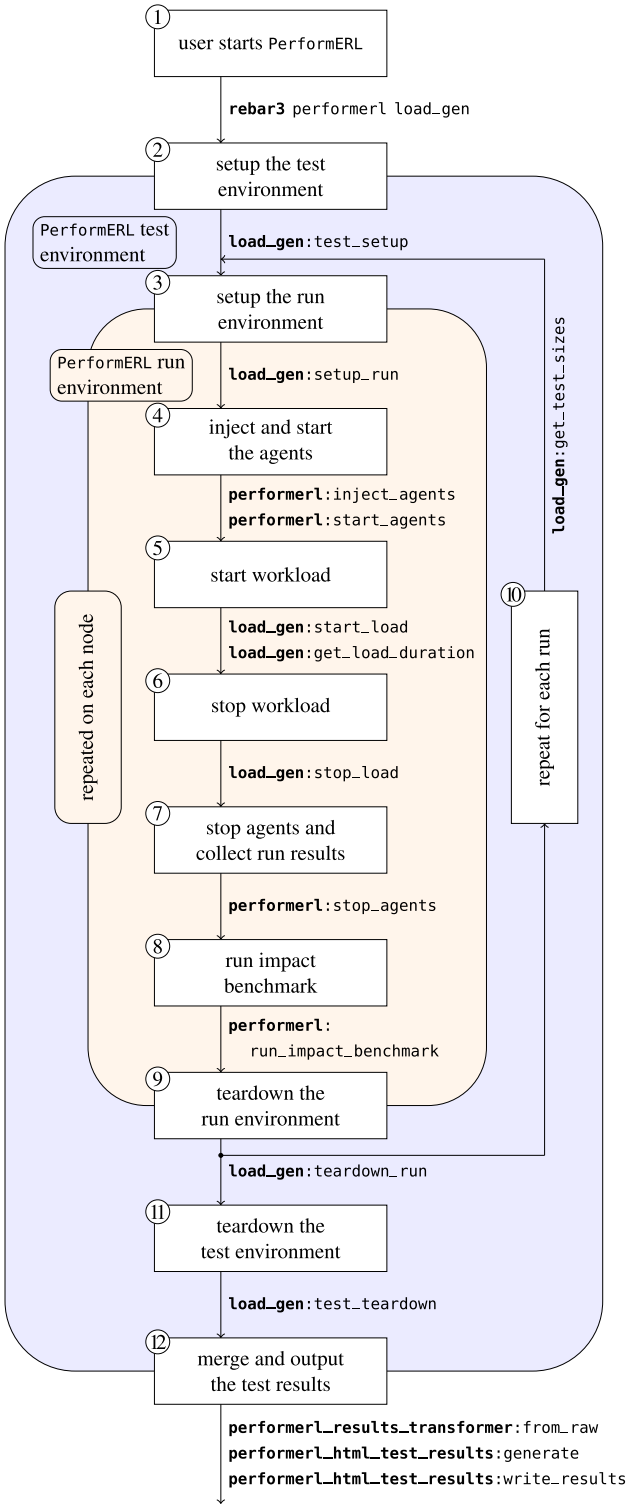


Fig. 1 PerformERL test execution flow

The target function patterns—a set of MFAs[1]—identify the group of functions of the SUT that the user is inter-

[1] An MFA is a tuple uniquely identifying an Erlang function through a module, a name and an arity.

ested in monitoring for a specific test case. These will be used as a starting point for the performance analysis made by PerformERL. By exploiting Erlang tracing infrastructure[2], PerformERL gathers data about the target functions themselves, most notably, the number of times they are called and their execution time. PerformERL also discovers any process in the SUT that makes use of the target functions and gathers metrics on those processes, including memory usage and reduction[3] count.

A PerformERL *test* starts when the user invokes the framework providing a load generator file. Since the goal of the performance test is to provide insights into the scalability of the SUT, every test is composed of multiple *runs*. Runs are successive calls to the same load generation functions, but with different values for the *size* parameter. The core task of each *run* is to exercise the SUT by generating a computation load—called *workload*—for the monitored application proportional to the given *size* parameter. Finally, when all the test runs have been completed, PerformERL produces its output as a collection of HTML files with charts and tables presenting the gathered results. Note that PerformERL does not target any specific scalability dimension but it aims to be flexible enough to allow the monitoring of any of them. The meaning of the *size* parameter depends on what the users would like to measure. For example, *size* can be the number of requests if we are interested in how the response time of a web server scales with the growth of the number of requests or it can be the number of entries in a database when we are interested in how the database size scales with the growth of the number of its entries. Figure 1 summarizes the details of a test execution flow in the PerformERL framework.

3 PerformErl under the hood

In this section, the different components of the framework will be described. Fig. 2 shows the components of PerformERL and how they interact with the test file provided by the user. In the following sections, white circled numbers—such as (step ①)—refer to steps of Fig. 1, whereas black circled numbers—such as (comp. ❶)—refer to components of Fig. 2.

3.1 The load generator behavior

The only file that users have to write in order to implement a test case is a *load generator*—i.e., a test file. The

```
-module(performerl_load_generator).

-type run_info() :: term().
-type test_info() :: term().
-type test_size() :: non_neg_integer().
-type trace_pattern() ::
   { module()|'_', atom()|'_', non_neg_integer()|'_' }.

-callback get_test_name() -> string().

-callback test_setup() -> {ok, test_info()}.

-callback setup_run(Size::test_size()) ->
    {run_started,[node()]}.

-callback start_load(
  TestNodes::[node()], Size::test_size()) ->
    {load_started, run_info()} |
    {already_started, pid()}.

-callback get_load_duration() -> pos_integer().

-callback get_test_sizes() -> {atom(), [test_size()]}.

-callback stop_load(RunInfo::run_info()) ->
    {load_stopped, run_info()} |
    {error, not_started}.

-callback teardown_run(RunInfo::run_info()) -> run_ended.

-callback test_teardown(TestInfo::test_info()) -> ok.

-callback get_trace_patterns() -> [trace_pattern()].
```

Listing 1: The load generator behavior

test files (comp. ❶) used by PerformERL must implement the performerl_load_generator behavior[4] shown in Listing 1.

To have different setup and tear down functions per test and per run enables the user to have more control over the generation of the test environment. The test_setup function is only called once at the beginning of the test (step ②). It can be used to start external services that are not directly involved in the performance test but are needed during the load generation steps or to perform operations that only need to be executed once during the test. In the run setup (step ③) and tear down (step ⑨), on the other hand, the user should take care of the actions that have to be done before and after each run, typically starting and stopping the SUT, so that every run will begin with the SUT in the same fresh state. The return value of the setup_run function must include the identifiers of the nodes in which the SUT is running; these nodes will be referred to as *test nodes* (comp. ❸).

The start_load function (step ⑤) contains the code to stress the SUT to obtain its performance data. The stop_load function (step ⑥) stops any long-running operation initiated by its counterpart. The get_test_name and get_trace_patterns functions are not explicitly used in Fig. 1 because they provide configuration parameters for a test, but do not affect the execution flow directly. They return the custom test name and the MFA of the target functions respectively.

[2] The BEAM provides a powerful set of tools for the introspection of events related to functions, processes and message passing that go by the name of Erlang tracing infrastructure.

[3] The reduction is a counter per process that is normally incremented by one for each function call.

[4] Erlang behaviors—as object oriented interfaces—define a set of callback functions that should be exported by any module implementing such behaviors. Failing to implement any of these callbacks generates a compiler warning.

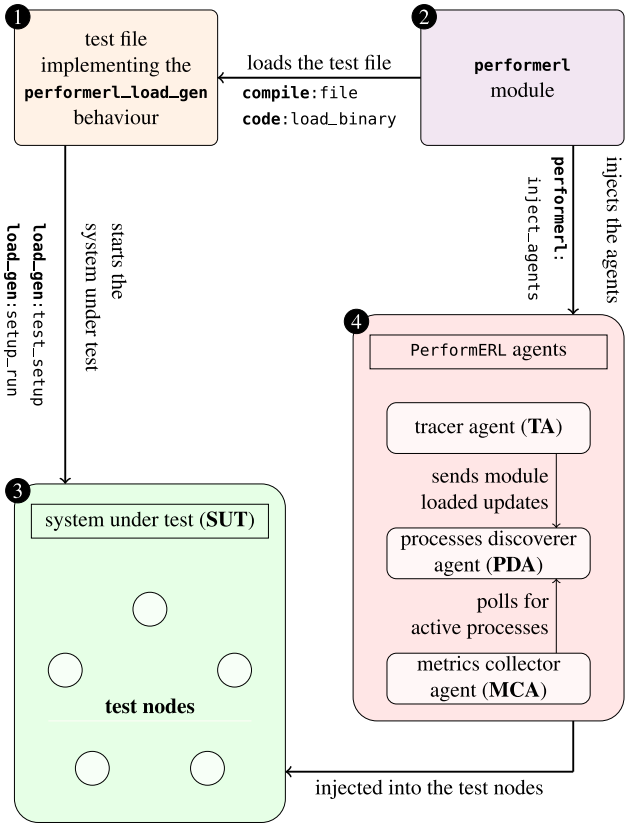


Fig. 2 PerformERL components interaction

The remaining test components are predefined in PerformERL and do not need to be customized by the user. In Sect. 4.2 we will discuss how PerformERL functionality can be extended.

3.2 The performERL module

The performerl module (comp. ❷) provides the entry point for every test execution. It contains a main function that loads the test file, sets up the global test environment common to all runs, and then starts a run for each user-specified size (step ⑩). Once all the runs have been completed, it takes care of tearing down the common environment and generates the output (steps ⑪ and ⑫).

The execution of a single run can be summarized in the following steps, also displayed in Fig. 1, where load_gen is the name of the test file provided by the user. First, the load_gen:setup_run callback is executed (step ③), which deploys the SUT on a set of Erlang nodes[5] (comp. ❸) whose identifiers are returned. The Erlang nodes are then instrumented by injecting the modules needed for the performance monitoring (step ④). The injected modules implement the *tracer* agent (TA), *processes discoverer* agent (PDA) and *metrics collector* agent (MCA) (comp. ❹): there will be an instance of each agent on every Erlang node. Once the agents are started, the function load_gen:start_load is called (step ⑤). PerformERL will wait for the load generation timeout to expire. The timeout is set by the function load_gen:get_load_duration, and its value must be large enough to enable the SUT to react to the generated load. Finally, data from the PDA and MCA will be gathered (step ⑦) and the run will be effectively ended. The only remaining step before cleaning up and stopping the test nodes is to execute the *impact benchmark* (step ⑧), and to use its results to refine the performance data.

3.3 The tracer agent

TA is the first agent started on the nodes running the SUT. The first purpose of this agent is to use *call time* tracing to measure the number of calls and the execution time of the target MFA patterns. *Call time* tracing, enabled by the trace flag call_time, is a feature of the Erlang tracing infrastructure that, for every traced MFA, records on a per process basis how many times the function was called and how much time was spent executing the function. Users can refer to this data with the function erlang:trace_info. *Call time* tracing does not require any message passing, as it only updates some counters maintained by the BEAM. The other purpose of TA is to interact with the Erlang tracing infrastructure and to track any process—apart from the PerformERL agents—that use the tracing primitives during the tests. By doing this, PerformERL is aware of the context in which the tests are executed and it can work, to a certain extent, even if the tracing infrastructure is already in use by the SUT. This is required to keep overheads under control, as the BEAM only allows one tracer agent per process.

In PerformERL, since it is unknown who will call the monitored functions, every process in the SUT has to be traced. This could be accomplished by the erlang:trace function, but to tolerate the need of a SUT to use the tracing infrastructure, PerformERL has to employ a more sophisticated approach: the Erlang meta-tracing facility. Meta-tracing is applied to an MFA pattern, and it traces the calls made by any process to the functions selected by such MFA pattern, without explicitly tracing the caller. To be bound to the MFAs enables a finer tracing mechanism that allows more tracer agents per process, making PerformERL tolerant to the presence of other tracers. Note that, a SUT using the tracing facility can be observed by PerformERL thanks to the adoption of the meta-tracer. But, a SUT that uses the meta-tracing facility can not be observed because one meta-tracer can be associated to one process.

[5] A Erlang node, node for short, is an instance of the BEAM. Several processes run on each node. Each process can communicate both with processes running on the same node and with processes running on other nodes even over the Internet.

Fortunately, this limitation has a negligible impact on the applicability of PerformERL because the meta-tracing facility is less frequently used than the standard tracing one.

The TA is started before the load_gen:start_load function is called, and sets itself as the tracer for all the other processes in the VM. The MFA patterns to trace are those specified by the user with the function load_gen:get_trace_patterns. Then TA sets itself as the meta-tracer for the tracing built-in functions[6] in order to detect if the SUT is making use of the tracing infrastructure and react accordingly. TA also sets itself as the meta-tracer for the erlang:load_module function, which is responsible for loading a module into the BEAM. This permits to monitor the calls to the functions described by the MFA patterns in dynamically loaded modules. These would otherwise be missed because the *call time* tracing feature is applied to the MFA patterns when TA is started. If this happens before the workload triggering the dynamic module loading, the dynamically loaded module containing the specific MFA would not be traced. In other words, TA can detect when a module containing some user-defined MFA patterns is being dynamically loaded and promptly activate *call time* tracing for those.

3.4 The processes discoverer agent

PDA tracks those processes that—at any point in their lifetime—use the monitored MFA patterns. PDA is started after TA and depends on it for the detection of newly loaded modules. PDA also uses the tracing infrastructure and it is where the most sophisticated tracing techniques are employed to quickly discover the processes with a low overhead.

The approach is simple: PDA is notified about a process presence with a tracing message, stores its PID and starts monitoring it as soon as it calls a function matching a user-defined MFA pattern. Then PDA immediately stops tracing the process to reduce the overhead on the SUT—details in Sect. 3.4.1. Notice that, because of the meta-tracing, the set of traced MFA patterns is limited to user-defined ones, but the space of traced processes is the whole BEAM runtime.

3.4.1 Match Specifications

The Erlang tracing primitive erlang:trace_pattern accepts as its second parameter an argument called *match specification*. Match specifications can be used to control and customize the tracing infrastructure. They are Erlang terms describing a low level program used to match patterns, execute logical operations and call a limited set of commands. Match specifications are compiled into intermediate code interpreted by the BEAM more efficiently than the corresponding function call.

PerformERL uses match specifications to limit the set of processes sending a message to the PDA to those who have not been discovered yet; this is equivalent to disabling the tracing facility for the other processes, reducing overheads. The list of known active processes is encoded as a balanced binary search tree sorted on the PIDs, translated into a match specification with short-circuited boolean operators. The list of active processes is kept updated by removing those that terminate their execution. The match specification is rebuilt whenever the list is updated. The cost of executing the match specification against the PID of a target function caller is logarithmic in the number of processes, because of the balancing of the binary tree structure.

3.4.2 The Custom Meta-Tracer

Meta-tracing is a powerful feature of the BEAM, but it is less customizable compared to regular tracing. With the regular tracing, the user can specify a number of flags to alter the format of the generated trace messages. These flags are unavailable when using meta-tracing. In particular, the arity flag—if available—would ease PDA implementation because it forces the trace message to contain the arity of the called function rather than the full list of its arguments. Since sending a message implies the copying of its data, sending trace messages containing only the number of arguments instead of the arguments themselves would significantly decrease the overhead of the meta-tracing.

Even though meta-tracing cannot be customized, it is possible to provide a tracer module when setting a meta-tracer. The tracing infrastructure allows the user to provide a custom module[7], composed of an Erlang stub and a NIF[8] implementation, to replace part of the back-end of the tracing infrastructure. It is therefore possible to code a custom tracer that implements the arity flag and further reduces the overhead. Ślaski and Turek [30] demonstrated the efficiency and potential of custom tracer modules.

3.5 The metrics collector agent

MCA is responsible for polling PDA for active processes and gathering metrics—e.g., memory usage and reductions count—about them. The metrics are collected by default every 5 seconds, but this interval can be customized.

[6] The built-in functions to access the Erlang tracing infrastructure are: **erlang**:trace and **erlang**:trace_pattern.

[7] http://erlang.org/doc/man/erl_tracer.html—Erlang tracer behavior.

[8] A NIF (Native Implemented Function) is a function written in C instead of Erlang. They appear as Erlang functions to the caller, since they can be found in an host Erlang module, but their code is compiled into a dynamically loadable shared object that has to be loaded at runtime by the host module.

The metrics collected by the MCA are sanitized to remove the tracing overhead from the *call time* data at the end of each run. The sanitation consists of removing the (average) overhead introduced by `PerformERL` tracing from the execution time of the monitored functions. `PerformERL` injects the *impact benchmark* module into the SUT when the run ends, when both the *call time* data and the number of discovered processes are available. This module measures the average overhead of tracing—due to both the *call time* and the processes discovery—on the monitored function calls. The impact benchmark executes the monitored function 4,000 times without any tracing enabled. Choosing 4,000 iterations ensures that the process will not exceed its time slot[9] and there will be no overhead due to context switching. The impact benchmark then spawns a number of processes equal to the highest number of active processes recorded during the run and activates both the *call time* tracing and the processes discovery meta-tracing with a match specification containing their PIDs. Each spawned process will execute the target function 4,000 times. The benchmark module concludes by taking the average execution time over all processes, subtracting the reference measurement to determine the impact of the tracing. Once the impact measurement is completed, the average latency multiplied by the number of calls is subtracted from the *call time* tracing data of each monitored function.

3.6 PerformERL in different ecosystems

`PerformERL`'s approach is designed for performance testing of Erlang's processes but it can be applied to any ecosystem supporting the actor model [2]. The actor model adopts a finer concurrency unit (the actors) than processes/threads whose implementation usually can not rely on the operating system mechanisms but requires an *ad hoc* virtual machine as the BEAM with its scheduling algorithms and communications mechanisms. Erlang natively supports the actor model—Erlang's processes are de facto actors—but there are several implementations for other ecosystems. The assumption on the concurrency model is not so stringent. It just simplifies `PerformERL`'s transposition.

Akka [17,18] is certainly the most relevant implementation of the actor model outside the Erlang ecosystem. Akka was born as part of the Scala standard library to then become the reference framework for all the languages supported by the JVM but also Javascript [32]. Akka is heavily inspired by Erlang and implements many of the fundamental actor model primitives offered by the BEAM, such as supervision trees and thread dispatchers (corresponding to Erlang schedulers). Moreover, Akka threads are not mapped to JVM threads, they are lightweight abstractions whose performances can be monitored by a dedicated framework such as `PerformERL`.

The test orchestration functionality of `PerformERL` can easily be reproduced in Akka since it provides all the necessary building blocks, such as nodes distribution, message passing, remote code injection and remote procedure calls. The only fundamental component that Akka and the JVM do not offer out-of-the-box is an equivalent of the Erlang tracing infrastructure, which `PerformERL`'s approach heavily relies on. `PerformERL`'s approach requires an interface to gather metrics for the `{actor, method}` pair and also to perform dynamic discovery of actors calling specific methods, without manually altering the source code of said methods. Ciołczyk *et al.* [12] describe Akka's limitations wrt. message exchange tracing. Tracing support comes from third-party libraries/frameworks. `Kamon`[10] provides metric gathering functionality for Akka actors, but lacks the possibility to send a message to an equivalent of the PDA. As demonstrated by the Akka tracing tool [12] and `AkkaProf` [28], the "last missing mile" can be realized via code instrumentation—either aspect-based [23,26] or based on bytecode instrumentation [8,13]. These techniques permit to weave/inject the tracing code to the tested methods (equivalent to `PerformERL` target MFAs), and to gather and pass information about the monitored method calls to the equivalent of the PDA to perform dynamic actor discovery.

4 `PerformERL` in action

`PerformERL` is available as a `rebar3` plugin and it is also compliant to the `escript` interface.

4.1 A usage example: `Wombat` Plugins

`Wombat` [33] is a performance monitoring framework for the BEAM. It works by injecting a number of *agents* into the monitored nodes to support its functionality. In this section, we will show how `PerformERL` can be used to measure the impact of the `Wombat` agents and their infrastructure on the managed nodes.

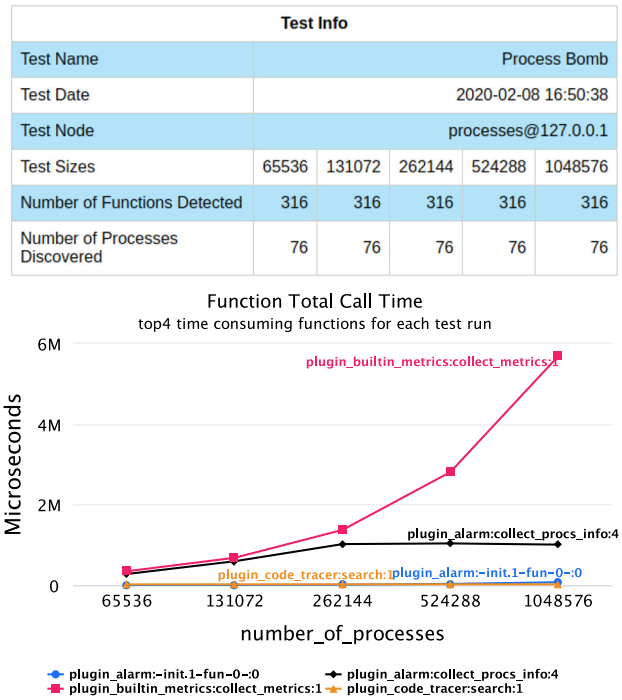
Listing 2 shows a portion of the `PerformERL` load generator file to exercise the `Wombat` agents by spawning a large number of processes in the monitored node. `PerformERL` monitors the processes and the function calls of `Wombat` when it is monitoring another SUT whose number of processes grows accordingly to the specification in the load generator file. As explained in Sect. 3.1, this load

[9] Erlang run-time system adopts a preemptive scheduler. Each process receives a *time slice* measured by a reduction count before being preempted, where a reduction is a function call. Since OTP20 the number of allowed reductions is 4,000.

[10] https://kamon.io/docs/latest/instrumentation/akka/

```
-module(processes_load_gen).
-behaviour(performerl_load_generator).
test_setup() ->
    ok = wombat_lib:ensure_wombat_started(),
    {ok, []}.
get_test_sizes() ->
    {number_of_processes,
     [65536,131072,262144,524288,1048576]}.
setup_run(Size) ->
    Node = 'processes@127.0.0.1',
    StartCmd = "erl -detached -name "++
               atom_to_list(Node)++
               " -setcookie "++
               atom_to_list(erlang:get_cookie())++
               " +P "++integer_to_list(Size),
    [] = os:cmd(StartCmd),
    % omitted: waiting for the node to come online
    ok = performerl_lib:inject_mod(?MODULE, Node),
    {run_started,[Node]}.
start_load([Node], Size) ->
    {node_added, WoNodeId} =
        wombat_lib:add_node_to_wombat(Node,
            atom_to_list(erlang:get_cookie())),
    Pids = rpc:call(Node,?MODULE,spawn_processes,[Size]),
    {load_started, [{node_info, {Node, WoNodeId}},
                    {pids, Pids}]}.
%********** RPC target function **********%
spawn_processes(Size) ->
    Num = (Size * 95) div 100,
    Pids = [spawn(fun() -> receive stop -> ok end end)
            || _ <- lists:seq(1, Num)].
```

Listing 2: PerformERL test file for testing Wombat plugins



Test Info

Test Name	Process Bomb				
Test Date	2020-02-08 16:50:38				
Test Node	processes@127.0.0.1				
Test Sizes	65536	131072	262144	524288	1048576
Number of Functions Detected	316	316	316	316	316
Number of Processes Discovered	76	76	76	76	76

Fig. 3 Results summary extract for the test in Listing 2

generator file implements the `load_generator` behavior to be used with PerformERL. The `test_setup` function checks that Wombat is up and running. The `setup_run` function spawns a node with a system limit for the number of processes set to the run size parameter. It injects the test module itself into the monitored node to enable the methods of the test file to be called on the test node, as it happens for the `spawn_processes` function. The `start_load` function adds the test node to Wombat and remotely calls—via the Erlang `rpm` module—its `spawn_processes` function. It will start a number of idle processes equal to the 95% of the processes system limit. The choice of 95% permits to have a meaningful load, close to the saturation threshold, and still to be sure not to reach it provoking the test crashing. The maximum number of processes can be retrieved by the call `erlang:system_info(process_limit)`. The `stop_load` and `teardown_run` functions (not shown in Listing 2) send a stop message to the spawned processes and take care of removing the test node from Wombat and shutting it down, respectively.

After all test runs have been completed, the user will find the output of the framework in a folder named `performerl_results` with a sub-folder for the test execution containing a `front_page.html` file with a summary result data for the test and more detailed files for each detected function and each discovered process. Figure 3 shows an excerpt of the summary page for the test load in Listing 2 . The table shows that PerformERL detects the same number of processes and functions for each run. This is correct because Wombat does not change. The graph, instead, shows that the number of processes that Wombat monitors affect the call time of Wombat's functions (information monitored by PerformERL). Then, we can conclude that the function draining more resources (execution time) when the number of processes monitored by Wombat grows is `plugin_builtin_metrics:collect_metrics` whereas all the other Wombat's functions have an execution time consumption independent of the number of processes or with a negligible growth. Therefore any attempt of optimizing the execution time of Wombat should affect such a function (that is part of the Wombat framework).

4.2 How to extend PerformERL

PerformERL has been designed to be extendable. In this section, we will show how to create custom agents and collect custom metrics that will cooperate with the default PerformERL components.

4.2.1 Collecting Custom Metrics

The first proposed extension consists of adding a new metric to those collected by the MCA. The function `performerl_mca:collect_metrics` combines the

```
-module(performerl_custom_agent).
% functions called locally in the PerformErl module
%*****************************************************
-callback get_config(TestNode::node(),
     LoadModule::module(),
     TestSize::test_size()) -> map().
-callback process_results(TestNode::node(),
     Results::term()) -> ProcessedResults::term().
% functions called remotely in the test nodes
%*****************************************************
-callback start(Config::map()) ->
     {ok, Pid::pid()} | ignore | {error, Error::term()}.
-callback get_results_and_stop() -> {ok, Results::term()}.
```

Listing 3: The Erlang behavior for custom agents

metrics collected across runs: each metric is a tuple `{metric_name, MetricValue}`. To support a new metric we have to add how it is calculated and its result to the list of the collected metrics. As an example, let us track the message queue length of the discovered processes. This is done by adding

```
QLen = erlang:process_info(Pid,
       message_queue_len)
```

to the `performerl_mca:collect_metrics` function `QLen` contains the result as `{message_queue_len, Value}` and it is returned together with the other collected metrics. Similarly, the `performerl_html_front_page` module—that would present the collected metrics to the user—will be accommodated to present the new metric as well.

To add a new metric to MCA can be realized by a callback whereas to accommodate the way it will be displayed has to be done by editing the functions in charge of the visualization because of the well-known problems about the automatic organization of data visualization. In spite of this, it should be evident that the effort required to add a custom metric is limited.

4.2.2 Adding a Custom Agent

Adding a custom agent is another extension which can be applied to `PerformERL`. The framework provides a specific behavior, `performerl_custom_agents`, allowing custom agents to be started. The agents are Erlang modules implementing the four callbacks defined by the behavior (Listing 3) .

The functions `get_config` and `process_results` are called locally by the `PerformERL` module to pass the parameters of the `start` function to the agent and processing the results after each run ends. The functions `start` and `get_results_and_stop` are called remotely in the test nodes via RPC from the `performerl` module and simply start and stop the custom agent retrieving the results.

The custom agents modules are provided to `PerformERL` as a comma separated list of paths with the command line argument `--custom_agents`. `PerformERL`, through the function `compile_custom_agents`, parses the comma separated list of module paths and compiles the agents with the `binary` option. This will not produce the `.beam` file, but a binary which is loaded in the local node and injected in the test nodes using the `inject_custom_agents` function. The start function of each custom agent loads the configuration parameters—taken from a combination of the test node name, the test file and the size of the current run—and remotely starts the agent in all test nodes with the appropriate configuration. The custom agents are started after the standard ones, allowing users to rely on the functionality and services offered by the standard `PerformERL` agents in their custom ones. For the same reason, they are stopped before the standard agents running on the same node.

As a proof-of-concept, we implement a custom agent that checks the health of the SUT during the performance tests by periodically monitoring some invariant properties. The custom `invariant_checker_agent` implementation is shown in Listing 4 . The agent implements both the `performerl_custom_agent` and the `gen_server` behaviors. The generic server functionality is used to implement the agents starting and stopping functions, as well as the periodic invariant checks. In the `get_config` function, the parameters for the agent—i.e., the list of invariants to check and the interval between two consecutive checks—are taken from the load generator test file that implements an additional function (`get_invariants`) to be used with this custom agent. The `process_results` function is where the data produced by the agent during the run will be analyzed. In the example, the data are processed by the `process_results0` function (not shown for brevity) and then the results of the invariants checks are printed to the console. Processed data are finally returned to the caller—the `performerl` module, that will include them in the run results. The test load file has to define and specify (via the `get_invariants` function) the invariants that `PerformERL` has to check. Listing 5 shows how a different set of invariants can be specified for each node involved in a test. The first function clause defines an invariant about the maximum size for the tables in the database of the first node and provides a threshold value proportional to the size of the current run. The second clause defines the invariants for the second node, in which a web server is executed: the first one is to check that the web server is online at all times, and the second one checks the length of the request queue against a threshold value. The last invariant is common to all nodes and sets a threshold value of 1GB for the entire node memory.

At the moment, there is no interface in `PerformERL` to automatically generate HTML files from the data produced

```
-module(invariant_checker_agent).
-behaviour(gen_server).
-behaviour(performerl_custom_agent).
% performerl_custom_agent callback
start(Config) ->
  InitState = #state{
      check_interval = maps:get(check_interval,Config),
      invariants = maps:get(invariants, Config)
  },
  gen_server:start({local,?MODULE},?MODULE,InitState,[]).
get_results_and_stop() ->
  {ok, ResHist} =
      gen_server:call(?MODULE, get_results, infinity),
  gen_server:stop(?MODULE),
  {ok, lists:reverse(ResHist)}.
get_config(TestNode, LoadModule, TestSize) ->
  #{ check_interval =>
       round(LoadModule:get_load_duration()/100),
     invariants =>
       LoadModule:get_invariants(TestNode,TestSize)}.
process_results(TestNode, ResHist) ->
  {TotalChecks,InvResList} = process_results0(ResHist),
  io:format("Invariants were tested ~p times on "
            "test node:~p~n", [TotalChecks, TestNode]),
  lists:foreach(
    fun({InvName,V}) ->
       io:format("  -invariant ~p was violated "
                 "~p times~n", [InvName, V])
    end, InvResList).
% gen_server callbacks
init(InitState=#state{check_interval = Interval}) ->
  erlang:send_after(Interval,self(),check_invariants),
  {ok, InitState}.
handle_call(get_results,_From,State) ->
  {reply, {ok, State#state.history}, State}.
handle_info(check_invariants,State#state{history=Hist}) ->
  Res = check_invariants(State#state.invariants),
  erlang:send_after(State#state.check_interval,
                    self(), check_invariants),
  {noreply, State#state{history=[Res|Hist]}};
%omitted: non relevant gen_server callbacks
% internal functions
check_invariants(InvList) ->
  lists:map(
    fun({Name, MFA, Op, DefValue}) ->
      {Name, check_invariant(MFA, Op, DefValue)}
    end, InvList).
check_invariant({M,F,Args}, Op, DefValue) ->
  Value = erlang:apply(M,F,Args),
  case test(Value, Op, DefValue) of
    true  -> ok;
    false -> {violated, Value}
  end.
test(A, '==', B) -> A == B;
test(A, '=<', B) -> A =< B;
test(A, '>=', B) -> A >= B.
```

Listing 4: Custom `invariant_checker_agent`

by the custom agents, so the users will need to manually modify the `performerl_html_front_page` in order to present it—similarly to what has been done for displaying the custom metrics in Sect. 4.2.1. Future developments of `PerformERL` will include an overhaul of the code generating the output HTML files to adopt a more modular approach and to offer a behavior with set of callbacks to support seamless extensions to the data visualization.

```
get_invariants('db_node@127.0.0.1', RunSize) ->
    [{db_tables_check,
        {dm_module, get_tables_size, [max]},
        '=<', 32*RunSize},
      get_memory_invariant()];
get_invariants('web_server_node@127.0.0.1', RunSize) ->
    [{web_server_online_check,
        {web_server, get_info, [status]},
        '==', 'up_and_running'},
     {web_server_queue_check,
        {web_server, get_info, [request_queue_length]},
        '=<', 100*RunSize},
      get_memory_invariant()].
get_memory_invariant() ->
    {memory_check,
        {erlang, memory, [total]},
        '=<', 1024*1024*1024}.
```

Listing 5: Examples of `get_invariants` functions

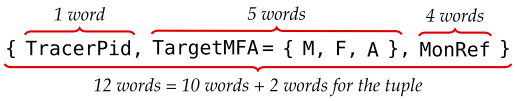
5 Evaluation

In this section, `PerformERL` overhead and performance are analyzed. All the presented tests were carried out on a 64-bit laptop equipped with an 8-core Intel i7@2.50GHz CPU and 8GB of RAM running Erlang/OTP version 20.3 on Linux. Any considered SUT has to run on the BEAM and can be composed of multiple types of nodes distributed across multiple machines.

5.1 Memory footprint of the agents

Running out of memory is one of the few ways in which an instance of the BEAM can be crashed. It is fundamental that the memory footprint of injected agents is minimal. In our evaluation, we calculate the upper bound of the `PerformERL` memory consumption. All reported calculations are based on the official *Erlang efficiency guide*[11], where the reference architecture is 64-bit and every memory word requires 8 bytes.

The TA does not keep any relevant data structures other than a list of processes using the Erlang tracing infrastructure alongside `PerformERL`. Every entry of this list contains a tuple of the form

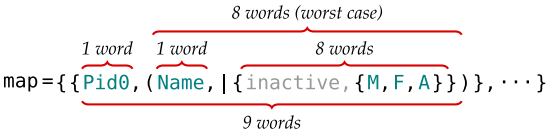


TA consumes 12 memory words for each traced MFA pattern plus 1 word for the pointer to the list and 1 word for each entry in the list. In total, TA consumes $13n + 1$ words of memory where n is the number of different traced MFA patterns. We can conclude that TA will never consume excessive amounts of memory, as it would require the SUT

[11] http://erlang.org/doc/efficiency_guide/advanced.html.

to trace about 10,000 different MFA patterns to consume a single MB of memory.

The PDA keeps two data structures: a `map` mapping PIDs to the names of the discovered processes and a `gb_sets` containing the PIDs of the active processes. The number of discovered processes is the upper bound for the active processes during the execution of a `PerformERL` test. The `map`



consumes 11 words per entry in the worst case. Its internal structure is implemented as a hash array mapped trie (HAMT) [6] when larger than 32 entries. According to the efficiency guide, a HAMT consumes $n \cdot f$ memory words plus the cost of all entries where n is the number of entries and f is a sparsity factor that can vary between 1.6 and 1.8 due to the probabilistic nature of the HAMT. Considering the worst case scenario, we have a total memory consumption of $1.8n + 9n = 10.8n$ words where n is the number of entries.

The `gb_sets` entries are the PIDs of the active processes which only take 1 memory word each. The data structure is based on general balanced trees [3] and is represented in Erlang by a tuple with two elements: the number of entries and a tree structure with nodes of the form `{Value, LeftChild, RightChild}` where the empty child is represented by the `nil`. The entire structure consumes 2 words for the outer tuple, 1 word for the number of entries, 3 words for the internal nodes with two children, 4 for those with only one child, and 5 words for the leaves. Since the `gb_sets` is a complete binary tree, if n is the number of entries there will be:

- $\lfloor (n+1)/2 \rfloor$ leaves (5 words each)
- $\lfloor n/2 \rfloor$ internal nodes (3 words each)

Note that at most one internal node has only one child (when n is even) by definition of complete binary tree. Summing it all up, the cost for the `gb_sets` of n active processes is:

$$3 + 5 * \lfloor (n+1)/2 \rfloor + 3 * \lfloor n/2 \rfloor + n \bmod 2$$

which is roughly $4n$ memory words.

The MCA holds in memory a list with the history of the collected metrics, so it naturally grows with time. Assuming, without loss of generality, that only the default memory and reduction metrics are collected, every entry of the list will be of the form

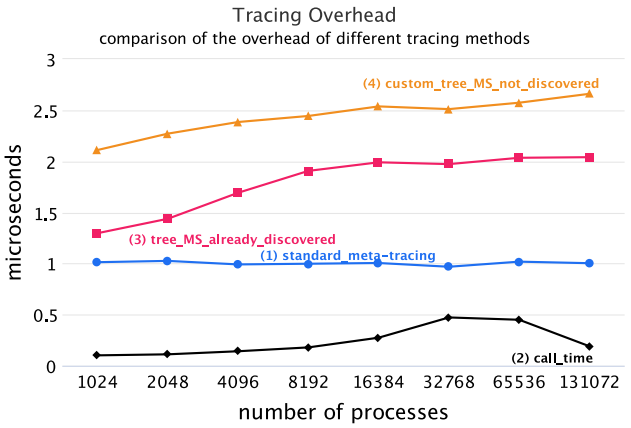
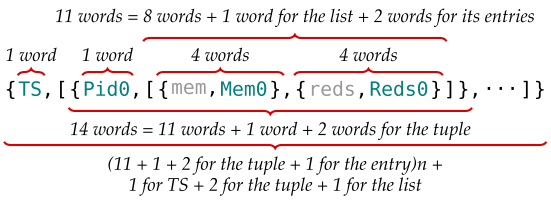


Fig. 4 Average overhead on a function call introduced by the tracing mechanisms. Four configurations for the tracing mechanism are considered: (1) meta-tracing only enabled (2) call time tracing, (3) meta-tracing with tree match specification and (4) meta-tracing with tree match specification and calling processes discovering. The x-axis (in logarithmic scale) reports number of processes spawned and the y-axis the average overhead over 4,000 calls to a dummy function



Such a list roughly consumes: $15n+4$ words where n is the number of active processes. To collect more metrics would only add the cost for their values representation. Assuming a standard 5 seconds interval between metrics collections, which gives 12 collections per minute, and 10,000 active processes, this structure will grow by approximately 1.8MB every minute.

The memory consumption of all the agents put together is roughly:

$$13p + 14.8d + (4 + 15d)c \sim \mathcal{O}(p + dc)$$

memory words, where

- p is the number of MFA patterns traced by the SUT
- d is the number of processes discovered (also used as an upper bound for the active processes)
- c is how many times the metrics are collected in a run (which depends on the duration and the metrics collection frequency).

It can be seen that the memory consumed by `PerformERL` agents is linear ($\mathcal{O}(p+dc)$) and depends on dimensions that the users can predict and control.

5.2 Overhead on monitored functions

In this section, we will explore the overhead PerformERL's agents add to the monitored functions. As explained in Sect. 3, the tracing of the MFA is the main culprit, if not unique, of the overhead introduced by PerformERL. Therefore, our experiment will focus on the average overhead PerformERL adds on the execution time of a dummy function called a fixed number of times (1,024 in our experiment) in an environment with an increasing number of processes (the *size* parameter) and with different tracing configurations activated. The idle processes—i.e., those that are not calling the dummy function—do not impact the call time directly but the resources used by the tracing infrastructure. It could seem counter-intuitive that we keep the number of the calls fixed when the total number of processes grows but this would permit to monitor how the tracing facility impacts the call time. The considered tracing configurations are:

1. meta-tracing enabled without any match specifications which always causes the caller to send a message. PerformERL does not use this configuration, but it provides a reference to compare with the other techniques;
2. *call time* tracing, which does not require any message exchanging but only updates some counters inside the BEAM. This is used by the TA.
3. meta-tracing with the tree match specification described in Sect. 3.4.1 and the calling processes identifiers already present in the tree. This case represents the already discovered processes calling a function and requires no message exchanging;
4. meta-tracing with the tree match specification and calling processes identifiers not present in the tree, so a trace message will be sent. This case represents the processes not yet discovered calling a monitored function. Messages are sent via the custom meta-tracer module described in Sect. 3.4.2.

For each tracing configuration and for each value of *size*, *size* processes are spawned enabling the tracing facility for a dummy function: the processes are assigned an ID from 0 to *size*-1. The spawned processes whose ID is a multiple of *size*/1,024 (i.e., ID = 0 mod (*size*/1, 024)) are selected to call the dummy function 4,000 times,[12] measuring the execution time with the timer:tc function. The average execution time for a single call to the dummy function is computed when all the selected processes have executed the benchmark. The overhead is determined by subtracting a reference value obtained executing the same benchmark without any tracing mechanism enabled. Figure 4 shows the results of the experiment using a logarithmic scale on the x-axis. It demonstrates that the increasing number of processes only affects the tracing techniques (3) and (4). It can also be seen that the growth is logarithmic, which confirms the theory behind the tree match specification presented in Sect. 3.4.1. The *call time* tracing configuration (2) also shows a slight overhead increase for larger numbers of processes. This is likely due to the performance of the data structures internally used by the BEAM to store the counters. The results show that the techniques employed for the process discovery cause an overhead that is between 1.5 and 2 times higher than a plain usage of meta-tracing but they allow PerformERL to prevent already discovered processes from sending trace messages and avoid flooding the PDA. The higher overhead is due to the execution time of the match specification and in the last configuration (4) also to the custom meta-tracer module being activated to send a custom message.

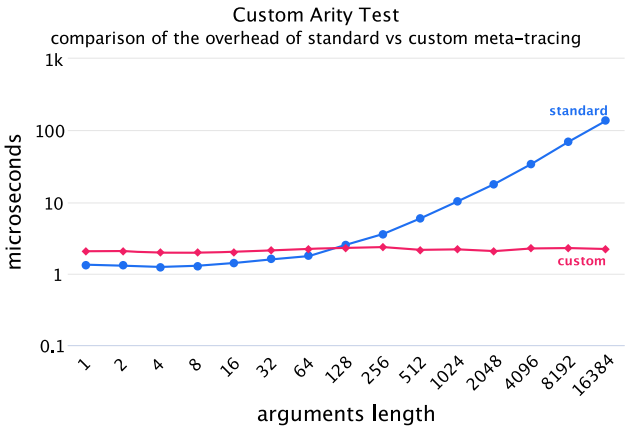


Fig. 5 Comparing the average overhead due to the standard meta-tracing and with the arity extension. The x-axis (in logarithmic scale) reports the number of integers passed to the calls. The y-axis reports the average overhead (in microseconds) over 100,000 calls to a dummy function

A second experiment has been done to show the importance of the custom meta-tracer module introduced in Sect. 3.4.2. This experiment compares the average overhead imposed by meta-tracing using the standard back-end (that sends a trace message containing the full list of arguments) with meta-tracing using PerformERL custom meta-tracer module implementing the arity flag. It measures the average execution time of a traced dummy function called 100,000 times for each configuration. Configurations differ for a parameter called *argument size* that determines the length of the list of integers passed to the dummy function. Since sending a message requires to copy the data to be sent, passing large parameters to a monitored function causes an increase in the tracing overhead.

Figure 5 presents the results of the tests. For small arguments, the custom meta-tracer causes a slightly higher

[12] The choice of 4,000 grants that the number of reductions will not trigger the scheduling algorithm avoiding the overhead due to the context switching.

overhead compared to the standard back-end because it needs to access a dynamically loaded shared library in addition to the BEAM tracing infrastructure. It can be seen that the overhead starts to diverge for arguments larger than a list of 64 integers: up to 100 times for a list of 16,384 integers which is not an unlikely size of arguments for an Erlang function call. In fact, the custom meta-tracer module acts as failsafe for the standard back-end when a process calls a monitored function with a very large argument. In this scenario, two undesirable things can occur: the process slows down due to the copying of the arguments and the PDA runs out of memory if too many of these messages are sent to it.

5.3 `PerformERL` in the real world

To show that the overhead introduced by `PerformERL` monitoring and tracing facility to the running SUT is average if not less than the one of the other frameworks, we measure it on a real case: `cowboy`[13], a well-known Erlang HTTP server, and compare it with the overhead of similar Erlang tools. In addition to `PerformERL`, the other chosen tools were `Wombat` [33], a proprietary performance monitoring framework, `eprof`[14] and `fprof`,[15] two profiling tools distributed with the Erlang standard library. Unfortunately, to the best of our knowledge, there are no other performance testing frameworks for the Erlang ecosystem and we have to compare `PerformERL` with performance monitoring frameworks. To maintain the comparison fair, we are measuring a resource (the average response time) observable without accessing to the SUT data structures: access that the performance monitoring framework do not have. The experiment will measure the server average response time to a number of HTTP GET requests both when the monitoring facility is active and when it is not.

The configuration of `PerformERL` used in this experiment had the target MFA patterns matching all the functions inside the `cowboy` codebase. `Wombat` was used with a standard configuration. `eprof` and `fprof` were set up to trace every process in the `cowboy` server node. For each tool, the experiment was set up with five increasing amounts of HTTP requests to measure the impact of the tools under different workloads. The requests are synchronous: a new request is made when the results of the previous one are received. In this way, each request is satisfied when cowboy receives it and no time is spent in a waiting queue that would bias the final measurements. For each of the described settings the experiment was run 100 times and the results of each set up were averaged to minimize any spike due to external factors beyond our control.

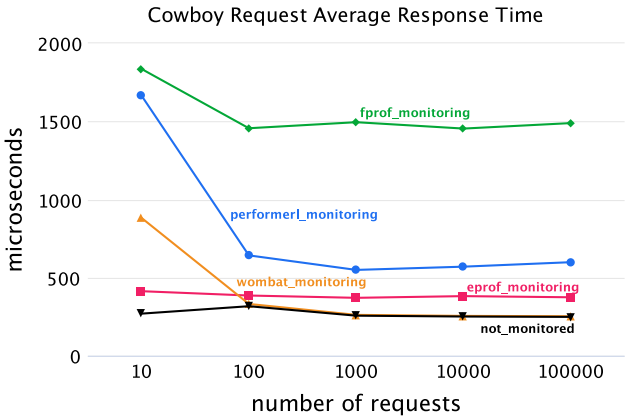


Fig. 6 Average overhead on `cowboy` response time

Figure 6 shows the results of the experiment, in terms of average response time for each workload. From the diagram, it can be noticed that all the considered tools caused a noticeable overhead when the number of requests is low. This is likely due to the tools performing some initial operations, such as setting up their monitoring facility, that affects the first few requests received by the server. In particular, `PerformERL` imposed a higher slowdown factor of 6 that settles on 2 with the growing of the workload. The initial peak can be attributed to `PerformERL` PDA which must discover all the `cowboy` processes, populate its data structures, and update the match specifications before the first request could be served. The cusp corresponds to when the number of requests is such that their satisfaction allows to mitigate the initial overhead. At that point the slowdown can be attributed to the heavy usage of tracing done by `PerformERL`, as described in Sect. 5.2. `eprof` shows a slowdown of around 1.4 for every workload. This tool is only employing *call time* tracing which, as shown in the previous section, causes a smaller overhead on the monitored functions, hence the slowdown factor is lower compared to `PerformERL` as expected. `Wombat`, similarly to `PerformERL`, causes a higher overhead in the monitored node in the first few seconds after its deployment due to the setting up of its plugin infrastructure. After that it can be seen that over time `Wombat` does not impose any overhead at all. `fprof` is the tool that showed the highest overhead in the experiment, with a constant average slowdown factor of 5 across all workloads. This is due to the heavy use of the tracing infrastructure done by `fprof` which traces every function call made by the monitored processes and writes the trace message in a file that will later be analyzed to produce a detailed call graph reporting for each function how much time was spent executing code local to the function and in functions called by that one.

[13] Cowboy—Small, fast, modern HTTP server for Erlang/OTP: https://github.com/ninenines/cowboy.

[14] https://erlang.org/doc/man/eprof.html.

[15] https://erlang.org/doc/man/fprof.html.

The experiment shows that the overhead caused in the web server by the monitoring tools is proportional to the usage of the tracing infrastructure, after an initial startup time where some tools, namely PerformERL and Wombat, have to setup their infrastructure which competes with the web server for the scheduling, causing an increase in the slowdown. The usage of the tracing infrastructure depends on the features that the tool offers regarding function calls. fprof provides more detailed information about function calls compared to the other tools and, for that reason, is the one with the highest overhead. PerformERL places in the experiment between eprof and fprof and in fact, it provides the same information as the former regarding function calls, but it also uses tracing for the real-time discovery of processes, which is a feature that no other tool offers. Wombat is different from the other tools since it is meant for monitoring of live production systems and focuses more on system wide metrics rather than function calls, so it can afford to limit the usage of the tracing infrastructure resulting in an overhead of almost zero, at least in a standard configuration.

5.4 Discussion

PerformERL should be included in the testing pipeline of a project and is not meant to be used in a production environment. This means that the primary goal of the framework is to provide a thorough insight into the SUT whilst offering compatibility with as many applications as possible, rather than achieving a low overhead. Nevertheless, the tests and estimates presented in this section show that the users can predict the dimension of the overhead caused by PerformERL, both in terms of memory consumption and execution time overhead. Both these dimensions depend on the number of processes that the SUT spawns and how many of them PerformERL has to discover.

In general, PDA and MCA provide useful information when there is a limited set of long-lived processes. On the other hand, trying to get information over a large number of worker processes that execute for a very short time before terminating will not provide any useful insight other than the number of such processes and the list of monitored functions that they called, whilst degrading the performance of the agents. This is a limitation inherent to the design of the PerformERL framework and the Erlang tracing infrastructure itself, also discussed by Ślaski and Turek [30]. To mitigate this issue, we are developing an extension to PerformERL that will enable the possibility of disabling some of the agents, at the cost of losing some of the standard features.

The real challenge PerformERL had to face is to apply performance testing on SUTs as Wombat [33] that actively need tracing to run without hindering their operation. In this respect, PerformERL uses and extends the meta-tracing facility to tolerate the use of the tracing infrastructure by the SUT (Sect. 3.3) as demonstrated by the experiment on Wombat reported in Sect. 4.1. Said that, PerformERL still has some limitations: for one, the SUT should not make use of meta-tracing. This is not an issue, as with existing Erlang applications, it seems that the meta-tracing facility is underrated and only used for troubleshooting live systems.

Another problem is the unloading (or reloading) of modules containing target function patterns during the tests. If this happens, the *call time* profiling data will be lost. In future versions, a periodic back-up of this data could be implemented at the cost of increased memory consumption, or a tracing-based mechanism monitoring the unloading and reloading of modules could be used to detect the issue.

6 Related work

The idea and the need of promoting performance testing in the early stages of the development cycle, which is one of the guiding principles behind this work, has been pointed out by Woodside *et al.* [35]. Others, such as Johnson *et al.* [22], suggested that performance testing should be incorporated in test driven development and that is indeed a goal that can be achieved using PerformERL.

In this section, we describe a few tools that are commonly used and share similar goals with PerformERL. The focus is on tools popular in the Erlang ecosystem but we will also discuss the most akin approaches even if unrelated to the BEAM.

6.1 Performance monitoring tools

In this paragraph we present tools related to PerformERL that fall in the category of *performance monitoring* tools in accordance to Jiang and Hassan [21] terminology.

A standard Erlang release ships with tools like eprof and fprof, that are built on top of the tracing infrastructure and provide information about function calls. A set of processes and modules to trace can be specified to limit the overhead, however, the approach of these tools is basic and has been improved in our framework to both reduce the impact on the SUT and gather more meaningful results. Furthermore, their output is text based, which may result in a poor user experience. More evolved tools, including PerformERL, process the output to generate reports with plots and charts to better help the user understand the gathered data.

Another tool already distributed with Erlang is the Observer[16]. Observer is an application that needs to be plugged into one or more running Erlang nodes offering a graphical user interface to display information about the sys-

[16] Observer, a GUI tool for observing an Erlang system: http://erlang.org/doc/man/observer.html.

tem such as application supervision trees, processes memory allocations and reductions, and ETS[17] tables. While some of the metrics gathered by this tool are similar to what PerformERL offers, the approach is different, as Observer is meant for live monitoring of entire nodes activity, whereas PerformERL is used to write repeatable tests and can focus on specific components of the SUT.

XProf [15] is a visual tracer and profiler focused on function calls and production safety. It achieves a low overhead by only allowing the user to measure one function at a time and gives detailed real-time information about the monitored function execution time, arguments and return value. Its purpose is mainly to be used to debug live production systems.

Wombat [33] is a monitoring, operations and performance framework for the BEAM. It is supposed to be plugged into a production system all the time and its features include gathering application- and VM-specific metrics and showing them in the GUI as well as sending threshold based alarms to the system maintainer so that issues and potential crashes can be prevented. The aim of Wombat is different from that of PerformERL, as it is not a testing tool, even if both share the idea of injecting agents into the monitored system to gather metrics.

Keiker [34] is a Wombat counterpart outside the BEAM written in Java. It replaces the Erlang tracing infrastructure by using aspect-oriented programming [23] to instrument code, but the users have to write the aspects, which requires to know AspectJ and an additional coding effort.

6.2 Load testing tools

In this section we present the related tools that—because of their black-box approach to performance testing—we categorize under the name of *load testing* tools, in accordance to Jiang and Hassan [21] terminology.

Apache JMeter [16] and Tsung[18] are widely used load testing tools. The former is written in Java and the latter is its Erlang counterpart. They share with our framework the repeatability of the tests and the idea of running them with increasing amounts of load but similarities stop there. Test configurations are specified via JSON-like files instead of code and their goal is to measure the performance of web applications—or various network protocols in general—under a large number of requests from an external point of view by looking at response times. PerformERL, on the other hand, provides information from the inside of the system, showing how each component reacts to the load.

Basho Bench[19] is a benchmarking tool created to conduct accurate and repeatable performance and stress tests inside the Erlang environment. It was originally implemented to test Riak [24] but can be extended by writing custom probes in the form of Erlang modules. The approach is indeed similar to the one used in PerformERL, but it focuses on two measures of performance—throughput and latency—related to network protocols and DB communications. Basho Bench differs from PerformERL in the sense that the former gives an overview of what the performance of an entire system looks like from the outside, while the latter provides insights into the performance of the system's components. Moreover, Basho Bench does not support the concept of run that permits to execute the same test with different loads. This is a crucial feature for a performance testing framework as PerformERL that must monitor how the SUT behaves as the load increases. Similar considerations can be done for BenchERL [4] as well.

Akka tracing tool [12] is a library to be used with Akka applications that permits to generate a trace graph of messages. It focuses on collecting metrics related to the messages exchange. It is extendable and provides an interfaces to show the collected data. It shares a philosophy and an architecture similar to PerformERL without providing its insights on the used resources/data structures. However, this is an extension whose support is envisionable since they already use AspectJ to inject the code to trace the messages (as we suggest in Sect. 3.6 for the PerformERL implementation on the JVM).

6.3 Performance testing tools

In this section we will present the tools related to PerformERL whose white-box approach we consider to be *performance testing*.

erlperf[20] is a collection of tools useful for Erlang profiling, tracing and memory analysis. It is mainly a *performance monitoring* tool but it offers a feature called *continuous benchmarking* meant for scalability and performance inspection that allows the user to repeatedly run tests and record benchmark code into test suites. This feature together with the collected profiling data suggest that erlperf could serve a purpose similar to PerformERL. However, the characteristics that would make erlperf a performance testing tool are still in a rudimentary state and no documentation is available to clearly understand their purpose and functionality.

detectEr tool suite [1,5] has some commonalities with PerformERL. They both target Erlang infrastructure,

[17] ETS tables are an efficient in-memory database included with the Erlang virtual machine.

[18] Tsung, a distributed load testing tool: http://tsung.erlang-projects.org.

[19] Basho benchhttps://github.com/basho/basho_bench.

[20] erlperf, a collection of tools useful for Erlang profiling, tracing and memory analysis: https://github.com/max-au/erlperf.

they both rely on the SUT execution for their analysis and both consider benchmarking and experiment reproducibility. Even if `detectEr` targets a post-deployment phase and runtime property validation. As `PerformERL`, `detectEr` relies on Erlang's actor model and the authors [1] discussed how the approach can be realized in other languages with different implementations of the actor model—with highlights similar to those described in Sect. 3.6. Due to its nature, `detectEr` has a limited view on the runtime usage of the resources. To some extends, the two approaches complement each other.

Stefan *et al.* [31] conducted a survey on unit testing performance in Java projects. From the survey, many tools emerged—such as `JUnitPerf`[21], `JMH`[22] and `JPL` [9]—that through various techniques apply microbenchmarking to portions of a Java application in the form of unit tests. This tools share with `PerformERL` the repeatability and a systematic approach aimed at testing performance, so we consider them *performance testing* frameworks. However, they are aimed at testing specific units of a Java system and mostly focus on execution time only.

A different approach to *performance testing* in Java was proposed by Bhattacharyya and Amza [7]. They proposed a tool, `PReT`, that tries to automatically detect any Java process in a system that is running a regression test and starts to collect metrics on them. The tool employs machine learning both to identify the processes running a specific test and to detect any anomalies in the collected measurements that could indicate a performance regression. The approach can definitely be considered *performance testing* but it differs from `PerformERL` in the sense that they evaluate performance measurements on tests already in place rather than providing an interface to generate a workload.

Moamen *et al.* [27] explored how to implement resource control in Akka-based actor systems. Their proposals share the general philosophy of `PerformERL` but are based on the manipulation of the basic mechanisms of the actor model: the spawning of the actors and the dispatch of the messages. The former permits to know the existence of an actor and then monitoring it since its spawning without the need of a PDA. The latter obviates to the need for a tracing facility. These approaches are more invasive and cannot be used to do performance testing of systems that cannot be stopped. `AkkaProf` [28,29] provides an approach similar to the one proposed by Moamen *et al.* [27] but instead of instrumenting the way an actor is spawned `AkkaProf` dynamically instruments the actors when their classes are loaded in the JVM. The injected code takes also care of collecting the metrics and sending them back to the `AkkaProf` logic agent (*de facto* implementing a sort of tracing facility).

7 Conclusion and future developments

This paper introduces `PerformERL`: a performance testing framework for the Erlang ecosystem. `PerformERL` can be used to monitor the performance of a SUT during its execution or be included in its testing pipeline thanks to `PerformERL` interface for defining load tests programmatically. `PerformERL` can collect several kind of metrics both about SUT internals and its behavior and it can also be extended with new metrics. Throughout this paper we have investigated `PerformERL` usability and visibility over the SUT, highlighted its flexibility demonstrating how it can be extended to match the user needs and the overhead it imposes over the SUT, showing both its strengths and weaknesses.

One of `PerformERL` weak points is the module used to visualize the results. Although it automatically shows the collected data, it is quite rigid wrt. the possible customizations of `PerformERL` forcing its manual extension to accommodate the visualization of new metrics. In future work, a more sophisticated approach could be adopted for the presentation of the test results that will ease the integration of data produced by both custom agents and custom metrics. Moreover, to increase the level of automation, future developments could include an interface to provide performance requirements—in the form of threshold values for the collected metrics—in order to define a pass/fail criteria [19]. Alternative criteria could be the no-worse-than-before principle defined by Huebner *et al.* [20] or the application of machine learning techniques as proposed by Malik *et al.* [25]. We are also considering to investigate how `PerformERL` could be integrated in the `detectEr` [5] tool chain.

Acknowledgments This work was partly supported by the MUR project "T-LADIES" (PRIN 2020TL3X8X). The authors wish also to thank the anonymous reviewers for their comments: they helped a lot in improving the quality of this work.

Funding Open access funding provided by Università degli Studi di Milano within the CRUI-CARE Agreement.

[21] https://github.com/clarkware/junitperf.

[22] Oracle Corporation, Java Microbenchmarking Harness: http://openjdk.java.net/projects/code-tools/jmh/.

References

1. Aceto, L., Attard, D. P., Francalanza, A., Ingólfsdóttir, A.: On Benchmarking for Concurrent Runtime Verification. In FASE'21, LNCS 12649, pp. 3–23, Luxembourg City, Luxembourg, (2021). Springer
2. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
3. Andersson, A.: General Balanced Trees. J Algorithms **30**(1), 1–18 (1999)
4. Aronis, S., Papaspyrou, N., Roukounaki, K., Sagonas, K., Tsiouris, Y., Venetis, I.E.: A Scalability Benchmark Suite for Erlang/OTP. In Erlang'12, pp. 33–42, Copenhagen, Denmark, (2012). ACM
5. Attard, D.P., Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Better Late Than Never or: Verifying Asynchronous Components at Runtime. In FORTE'21, LNCS 12719, pp. 207–225, Valletta, Malta, (2021). Springer
6. Bagwell, P.: Ideal Hash Trees. Technical report, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland (2001)
7. Bhattacharyya, A., Amza, C.: PReT: A Tool for Automatic Phase-Based Regression Testing. In CloudCom'18, pp. 284–289, Nicosia, Cyprus, (2018). IEEE
8. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A Code Manipulation Tool to Implement Adaptable Systems. In: Adaptable and Extensible Component Systems, (2002)
9. Bulej, L., Bureš, T., Horký, V., Kotrč, J., Marek, L., Trojánek, T., Tůma, P.: Unit Testing Performance with Stochastic Performance Logic. Automated Softw. Eng. **24**, 139–187 (2017)
10. Cesarini, F., Thompson, S.J.: Erlang Programming: A Concurrent Approach to Software Development. O'Reilly, (2009)
11. Cesarini, F., Vinoski, S.: Designing for Scalability with Erlang/OTP: Implementing Robust, Fault-Tolerant Systems. O'Really Media, (2016)
12. Ciołczyk, M., Wojakowski, M., Malawski, M.: Tracing of Large-Scale Actor Systems. Concurrency and Computation-Practice and Experience **30**(22), e4637 (2018)
13. Dahm, M.: Byte Code Engineering. In Java-Informations-Tage, 267–277, (1999)
14. Gheorghiu, G.: Performance vs. Load vs. Stress Testing [Online]. http://agiletesting.blogspot.com/2005/02/performance-vs-load-vs-stress-testing.html, (2005)
15. Gömöri, P.: Profiling and Tracing for All with Xprof. In: Proceedings of the Elixir Workshop London, London, United Kingdom, (2017)
16. Halili, E.H.: Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites. Packt Publishing, (2008)
17. Haller, P.: On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective. In AGERE!'12', pp. 1–6. ACM, (2012)
18. Haller, P., Odersky, M.: Scala Actors: Unifying Thread-Based and Event-Based Programming. Theoret. Comput. Sci. **410**(2–3), 202–220 (2009)
19. Ho, C.-W., Williams, L.A., Antón, A.I.: Improving Performance Requirements Specifications from Field Failure Reports. In RE'07, pp. 79–88, New Delhi, (2007). IEEE
20. Huebner, F., Meier-Hellstern, K., Reeser, P.: Performance Testing for IP Services and Systems. In GWPSED'00, LNCS 2047, pp. 283–299, Darmstadt, Germany, (2000). Springer
21. Jiang, Z.M., Hassan, A.E.: A Survey on Load Testing of Large-Scale Software Systems. IEEE Trans. Softw. Eng. **41**(11), 1091–1118 (2015)
22. Johnson, M.J., Ho, C.-W., Maximilien, E.M., Williams, L.: Incorporate Performance Testing in Test-Driven Development. IEEE Software **24**(3), 67–73 (2007)
23. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, B.: An Overview of AspectJ. In ECOOP'01, LNCS 2072, pp. 327–353, Budapest, Hungary, (2001). Springer-Verlag
24. Klophaus, R.: Riak Core: Building Distributed Applications without Shared State. In CUFP'10, pp. 14:1–14:1, Baltimore, Maryland, USA, (2010). ACM
25. Malik, H., Hemmati, H., Hassan, A.E.: Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems. In ICSE'13, pp. 1012–1021, San Francisco, CA, USA, (2013). IEEE
26. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: A Domain-specific Language for Bytecode Instrumentation. In AOSD'12, pages 239–250, Potsdam Germany, (2012). ACM
27. Moamen, A.A., Wang, D., Jamali, N.: Approaching Actor-Level Resource Control for Akka. In JSSPP'18, LNCS 11332, pp. 127–146, Vancouver, BC, Canada, (2018). Springer
28. Rosà, A., Chen, L.Y., Binder, W.: AkkaProf: A Profiler for Akka Actors in Parallel and Distributed Applications. In APLAS'16, LNCS 10017, pp. 139–147, Hanoi, Vietnam, (2016). Springer
29. Rosà, A., Chen, L.Y., Binder, W.: Profiling Actor Utilization and Communication in Akka. In Erlang'16, pp. 24–32, Nara, Japan, (2016). ACM
30. Ślaski, M., Turek, W.: Towards Online Profiling of Erlang Systems. In ERLANG'19, pages 13–17, Berlin, Germany, (2019). ACM
31. Stefan, P., Horký, V., Bulej, L., Tůma, P.: Unit Testing Performance in Java Projects: Are We There Yet? In ICPE'17, pp. 401–412, L'Aquila, Italy, (2017). ACM
32. Stivan, G., Peruffo, A., Haller, P.: Akka.js: Towards a Portable Actor Runtime Environment. In AGERE!'15, pp. 57–64, Pittsburgh, PA, USA, (2015). ACM
33. Trinder, P., Chechina, N., Papaspyrous, N., Sagonas, K., Thompson, S.J., Adams, S., Aronis, S., Baker, R., Bihari, E., Boudeville, O., Cesarini, F., Di Stefano, M., Eriksson, S., Fördős, V., Ghaffari, A., Giantsios, A., Green, R., Hoch, C., Klaftenegger, D., Li, H., Lundin, K., MacKenzie, K., Roukounaki, K., Tsiouris, Y., Winblad, K.: Scaling Reliably: Improving the Scalability of the Erlang Distributed Actor Platform. ACM Trans. Prog. Lang. Syst. **39**(4), 17:1-17:46 (2017)
34. van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In ICPE'12, pp. 247–248, Boston, MA, USA, (2012). ACM
35. Woodside, M., Franks, G., Petriu, D.C.: The Future of Software Performance Engineering. In FOSE'07, pp. 171–187, Minneapolis, MN, USA, (2007). IEEE