Profiling an Erlang Program Inside A Linux Environment: A Discussion of Possible Approaches

João C. Fukuda, Emilio Francesquini, Daniel Cordeiro

[1]Escola de Artes, Ciências e Humanidades – Universidade de São Paulo (USP)
{joao.fukuda, daniel.cordeiro}@usp.br

[2]Centro de Matemática, Computação e Cognição – Universidade Federal do ABC (UFABC)
e.francesquini@ufabc.edu.br

Abstract. *Erlang is a concurrent language built to run on the BEAM virtual machine. This paper discusses the different approaches one can take to profile programs built on Erlang from both inside and outside BEAM and their inherent tradeoffs. It divides profiling and tracing tools into system and VM-level, compares both categories' advantages and disadvantages in terms of performance and how trying to keep performance might minimize other aspects of the profiling, and describes how to take advantage of them both alone and together by combining their outputs to produce the best possible result.*

1. Introduction

As Erlang [1] runs on a virtual machine (BEAM), different profiling methods than the ones used on compiled languages might be needed. Suppose we try to acquire information about the running program from outside that virtual machine environment. In that case, some pieces of information might be inaccessible, or the tools will not be equipped to gather them. On this note, there is still use for the more traditional profiling tools, even on such programs, either because of performance issues with VM-level profilers or if the user needs a more methodical, careful, and low-level point of view.

Current preferred methods of profiling mainly involve VM-level tools and libraries [4, 6]. However, profiling outside the VM has also been considered both on Erlang [3] and other virtual execution environments [2, 5].

This paper intends to discuss the possible approaches one can take to better use all available tools for profiling on both VM and system-level. We present both approaches' advantages and disadvantages and try to work out ways to use them together. We also list and compare multiple pieces of software and some of their functionalities.

2. Profiling Tools

There are two possible approaches to Erlang profiling: either profiling from inside or outside the virtual machine. The use case and applications for both approaches depend heavily on memory and CPU usage limitations, Linux Kernel Modules, Erlang's source code, and BEAM's binary knowledge.

Erlang currently ships with a few different profiling and tracing libraries like fprof and erl_tracer that might or might not be helpful depending on the level of detail needed and possible hardware limitations. This is better discussed in Section 2.1.

Aside from built-in libraries, there are also library wrappers and standalone modules (some of which are even recommended by the Erlang documentation about profiling) that come with extra features, including data gathering capabilities, data analysis tools, information visualization methods, and output formatting.

All of these modules and wrappers take advantage of being inside of the virtual machine and do what most tools from outside cannot: they record numerous pieces of

Table 1. Comparison of VM-level Profiling Tools

Tool Name	Built-in Library	Call Graph	Outputs to CallGrind Format	Function Execution Time	Flame Graph
fprof	X	X	X	X	
looking_glass		X	X	X	X
recon		X		X	
eflame		X		X	X

information such as call graph, a function's module, function call count, CPU time use per function, relative and total function call time, process spawn count, among others.

Erlang also has methods to expose such pieces of information to system-level tools through `EPMD` (`Erlang Port Mapper Daemon`), allowing other profiling tools to work without sharing `BEAM`'s resources while still having access to these pieces of information. One such tool is `Erlyberly`.

Besides all approaches mentioned above, some approaches that are more "traditional" might also be considered. Approaches involving tracing, analyzing system and library calls on both kernel and userland coming from the running virtual machine.

Most of these profilers work with binary file symbols, which requires the user to recompile Erlang in debug mode and look at its source code for the functions related to these symbols (what they do and how they interact with the underlying Erlang code). Some profilers work by setting hooks from inside its source code or creating a kernel module to gather such information, which might imply the need for knowledge in other areas that are not Erlang related. It becomes more comfortable with time and practice, but it has a much steeper learning curve than other discussed profiling methods.

2.1. Profiling from Inside Erlang's Virtual Machine

Erlang profilers that work from inside `BEAM` are, for the most part, Erlang's built-in profiling libraries or their wrapper with some extra functionalities involving data parsing and gathering, and information inference. However, they all have a similar interface. These profilers also sometimes come at an overhead cost higher than what one can afford.

One of the biggest concerns involving VM-level profilers is the choice between performance and level of detail. Having both is, in most cases, not possible due to either memory or scope limitations.

Profilers work by first defining the tracing scope (modules, functions, and arities). Tracing then starts, stops, and has its data saved to disk, analyzed, and then parsed to another file where it is then usable by the end-user. The output is either meant to be used raw or with external tools like call graph visualizers. Some raw outputs might even have programs that convert this raw data to other output formats for better visualization.

Data retained in memory is a concern, for if there is not enough space, then the program will not be able to run alongside the profiler. The act of saving everything to memory and later writing to disk is the cause of many of the "not enough memory" errors and where the choice to profile a smaller scope comes into play. The user will need to know how and what to prioritize when choosing the profiler and defining the scope.

One last concern is the output format of these files. Many profilers address these problems, but the built-in libraries have no way to—by themselves—produce an easy-to-use output. There is a solution: most libraries' outputs can be either utilized raw or parsed into usable formats to work with other profiling tools. One such example is `Erlgrind` which parses Erlang's fprof outputs into `KCacheGrind` format for easier visualization.

Table 2. Comparison of System-level Profiling Tools

Tool Name	Plug-and-play	Requires Erlang Source Code Manipulation	Requires Building Linux Kernel Module	Saves output	Gathers info from inside `BEAM`
OProfiler	X			X	
Linux Perf	X			X	
SystemTap			X	X	
LTTng		X		X	
Erlyberly	X				X

The next step to problem identification comes from the insights gained from such profiling, as call graphs, time measurements, and all other valuable information help tracking and dealing with the identification. The tracing data gathered on spawned processes, user-made functions, and modules, is an excellent way of narrowing down possible optimization paths inside the codebase.

2.2. From Outside Erlang's VM

Erlang profilers from outside have many benefits regarding scope but lack detail in information from inside `BEAM`. They are the inverse of VM-level profilers, both performance and detail-wise.

One of the most significant downsides of most profilers from outside `BEAM` is the preparation and the amount of effort needed in order to gather and to use those pieces of information. Both require a fair amount of knowledge on Linux and Erlang's source code (but not so much about the compilation process as there are tools such as `Kerl`).

As most profilers rely on executable symbols to display data, it is of interest for the user profiling the program to recompile Erlang in debug mode to include `BEAM`'s symbols. However, keep in mind that these symbols are just `BEAM`'s way of communicating with the operating system; they do not represent user-created functions or the program's structure. The modules and function calls inside the VM are not known by system-level profilers as they are not capable of gathering these pieces of information through symbols.

Once with the information given by the profiler, the user can put it together with Erlang's source code by searching the implementation of these symbols to comprehend what they are responsible for in `BEAM` and, lastly, infer which parts of the program being profiled use these symbols' functions. It is a much more laborious process than using VM-level profilers, but it allows much broader profiling and pays off in the end.

Resources-wise, the system-level profilers are the best choice as they seem to use much less of the system's resources while profiling. They do a lot more disk writing in the process but reduces the memory usage problem that most internal profilers seem to have. Here, the problem might be the resulting disk usage.

The sheer volume of information gathered can be overwhelming, especially if using with some advanced options. For example, while a '`perf record`' command profiling a simulation taking about 50 seconds ran without any options outputs a file of about 15MB in size, running the same program using the call graph option ('`--call-graph dwarf`') on the same program might output a file of about 3.2GB in size.

`Erlyberly` works differently from other system-level profiling tools. It uses `EPMD`. By doing so, it has access to VM-level information even though it runs as a separate process outside `BEAM`. `Erlyberly` is meant to be a "plug and play" experience, but it might require some extra configurations if the user intends to differ from the common usage of `EPMD`. One such example is running `EPMD` on a different port from the default 4369 or inside a Docker container.

`Erlyberly`'s profiling session is not writable to disk. It resembles the use case of a process manager (such as `htop`). The user manually chooses an available running Erlang process to observe and can then choose to select individual modules and functions to monitor. `Erlyberly` will log all their actions, but only while on a session.

Due to its different nature, this profiler works best on server-like programs where it keeps running in the background and occasionally (upon user interaction) performs an action. Programs that automatically start and end (like a simulation that reads its configurations from files) might need to look elsewhere for profiling methods.

Using system-level profilers is best done by slowly building up the program's understanding of how it runs to infer or acquire new profiling information based on this new understanding. Table 2 compares all evaluated system-level profilers.

3. Conclusion

Understanding the different usage cases and having the knowledge to work with them together, as well as which information each can gather, will be best. For example, people looking for local optimization opportunities will benefit most from VM-level profilers as they will often identify and return only specific parts of code to optimize.

For a more thorough job: system-level profilers are best. With them comes a broader understanding of Erlang's virtual machine, the feel of knowing which components to use and which to avoid. The user begins to consider the costly programming logic and how the virtual machine will interpret and execute these instructions.

Although it has more benefits in the long run, these pieces of information are more focused on a broader and more fundamental understanding that affects coding style and choices as a whole; it might not point to a single piece of code but to a poor choice that has been made throughout the entire codebase. It requires more dedication and time to transform information into performance.

References

[1] Joe Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2013.

[2] Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009.

[3] Scott Lystig Fritchie. DTrace and Erlang: a new beginning. In *Erlang User Conf.*, 2011.

[4] Sebasthian Karlsson. Exploring the Elixir ecosystem testing, benchmarking and profiling. `https://odr.chalmers.se/handle/20.500.12380/219742`, 2015. Access: 13-04-2021.

[5] Andrea Rosà, Lydia Y Chen, and Walter Binder. Actor profiling in virtual execution environments. In *Procs. of the 2016 ACM SIGPLAN Intl. Conference on Generative Programming: Concepts and Experiences*, pages 36–46, 2016.

[6] Michał Ślaski and Wojciech Turek. Towards online profiling of Erlang systems. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang*, pages 13–17, 2019.