Energy Efficiency in Actor Systems: A Comparative Study of Akka and Elixir

Irina Samus
IMT Atlantique
Nantes, France
Vrije Universiteit Brussel
Brussels, Belgium
irina.samus@imt-atlantique.fr

Mario Südholt
IMT Atlantique
Nantes, France
mario.sudholt@imt-atlantique.fr

Coen De Roover
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

Jens Nicolay
Vrije Universiteit Brussel
Brussels, Belgium
jens.nicolay@vub.be

Abstract

The energy consumption of digital infrastructures and applications constitutes a critical global concern. To build a sustainable digital future, software engineering must prioritize energy efficiency alongside traditional performance metrics. Actors provide an attractive model for distributed large-scale applications; however, the energy consumption of actor systems has barely been explored.

This paper presents a comparative energy benchmarking study of two prominent actor-based frameworks: Akka, running on the Java Virtual Machine, and Elixir, running on the Erlang Virtual Machine. Utilizing high-resolution physical wattmeters on the Grid'5000 Cloud infrastructure, we evaluate these frameworks across single-node and multi-node distributed configurations. We used three benchmarks from the Savina benchmark suite to investigate the energy costs of actor spawning, message passing, and numeric computation.

We demonstrate that Elixir is significantly more energy-efficient for actor-managing, i.e., orchestration-heavy tasks, consuming more than 8x less energy for actor spawning and for message passing compared to Akka. Akka, in contrast, demonstrates superior energy efficiency for CPU-intensive numeric workloads.

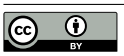
CCS Concepts

• **Computer systems organization → Distributed architectures**.

Keywords

Energy Benchmarking, Actor Model, Akka, Elixir, Distributed Systems

ACM Reference Format:
Irina Samus, Mario Südholt, Coen De Roover, and Jens Nicolay. 2026. Energy Efficiency in Actor Systems: A Comparative Study of Akka and Elixir. In *Workshop Proceedings of the 55th International Conference on Parallel Processing (ICPP Workshops '26), September 28–October 01, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3816891.3834893



This work is licensed under a Creative Commons Attribution 4.0 International License.
ICPP Workshops '26, Singapore, Singapore
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2763-4/26/09
https://doi.org/10.1145/3816891.3834893

1 Introduction

Hardware infrastructures, notably Cloud/AI infrastructures, as well as software infrastructures and applications today account for a significant share of the global energy consumption. Measuring the consumption of software artifacts is an important building block for improving their energy efficiency.

The actor model — featuring autonomous actors that can be deployed on different hosts, execute autonomously on those hosts and exchange messages with one another — constitutes, using different framework-based and language-based implementation strategies, a major building block for distributed large-scale applications. It is therefore important to precisely gauge its energy consumption. However, this question has only been rarely investigated until now.

In this paper, we provide an additional answer to this question by comparing two popular frameworks that implement the actor model: Akka and Elixir. We evaluate how their underlying architecture impacts energy consumption in a sequential and distributed environments. They run on very different virtual machines: the Java Virtual Machine (JVM) for Akka and the Erlang Virtual Machine (BEAM) for Elixir.

Benchmarking the energy efficiency of these systems is an important aspect of modern software engineering, as the choice of framework can significantly influence the energy cost of an application. Historically, software for distributed systems has been developed with a focus on ease of scaling, often overlooking the physical power costs of orchestration and communication. As global energy prices and environmental regulations become more stringent, understanding the trade-off between heavy numeric computation and lightweight message passing becomes essential. Thus, we hope that by comparing these two kinds of system activity, we can establish which scenarios are more energy-efficient for Elixir's lightweight process model and Akka's highly optimized execution environment.

This work was motivated by the observation that most of the research on actor frameworks focuses on performance metrics within a single machine or local shared-memory environment. While these

studies provide valuable insights into throughput and latency, they often overlook energy consumption in distributed, multi-node scenarios. In our research, we adapted several benchmarks from the standard Savina suite to function in a distributed setting, allowing us to investigate how actor systems behave when communication occurs across physical network boundaries. To achieve high-fidelity results, we conducted our experiments on the Grid'5000 Cloud infrastructure that allowed us to move away from software-based power estimation and instead use physical wattmeters to capture the real energy footprint of the frameworks under various distributed workloads.

In this paper, we analyze the energy performance of Akka and Elixir using three specific concurrency benchmarks. The first is a Spawn benchmark, which we used to measure the energy overhead of creating and destroying large numbers of actors, particularly focusing on memory allocation and garbage collection costs. The second is a Message passing benchmark, designed to evaluate the energy efficiency of asynchronous communication between actors across different physical nodes. The third and final benchmark is the Trapezoid approximation method for numeric integration; we used this CPU-intensive task to compare how each framework handles heavy mathematical computation and to investigate the impact of the JVM's Just-In-Time (JIT) compiler on long-term energy amortization.

In this work we provide the following contributions:

- We present the first study of a direct energy comparison between the Akka and Elixir actor models, in particular measuring their consumption within the JVM and BEAM virtual machines.
- We present benchmarks of single machine, concurrent computations and real-world distributed computations using physical wattmeters on the distributed Grid'5000 Cloud infrastructure.
- Our results show that Elixir is more energy-efficient for actor-managing, i.e., orchestration, primitives, such as spawning and messaging, while Akka is significantly more efficient for heavy numeric computations.

The remainder of this paper is organized as follows. Section 2 provides the background and discusses related work in actor performance and energy profiling. Section 3 details the experimental setup, including the hardware used on Grid'5000 and the implementation of the distributed Savina benchmarks. Section 4 presents the results of our experiments and provides a discussion of the findings, specifically regarding the trade-off between orchestration and computation. Finally, we conclude the paper and suggest directions for future work in Sec. 5.

2 Background

The actor model, introduced by Carl Hewitt [8] in 1973, proposes actors as the universal primitive of concurrent computation. In this model, an actor is an independent computational entity that encapsulates its own state and communicates exclusively through asynchronous message passing. A defining characteristic of the model is the absolute absence of shared memory, which inherently prevents data races and simplifies the development of complex parallel systems. Actors process messages one at a time from a private mailbox, allowing for high levels of concurrency and fault tolerance. Furthermore, the model provides "location transparency", meaning the semantics of message passing remain the same whether an actor is located on the same processor or on a different node in a distributed network. [5]

2.1 Akka on the JVM

Akka was introduced in 2009 by Jonas Bonér as a response to the challenges of scaling Java-based systems in a multi-core and distributed world. Inspired by Erlang's reliability, the core design goal was to bring the "Let it Crash" philosophy to the Java Virtual Machine. Unlike frameworks that treat concurrency as an add-on, Akka implements the actor model as a fundamental middleware layer that abstracts away the complexities of manual thread management and locking [15].

In the Akka architecture, an actor is a stateful object that is strictly encapsulated and its internal state is inaccessible to other actors except through asynchronous messages. Actors are organized into a strict hierarchy within an *ActorSystem* where every actor is a "parent" to the sub-actors it creates, forming a supervision tree. This hierarchical structure is central to Akka's fault tolerance: parents are responsible for deciding whether to restart, stop, or resume a child actor upon failure. Technically, Akka actors are highly decoupled from the underlying hardware threads as they are managed by *Dispatchers*, which utilize efficient fork-join thread pools to schedule actor execution. Because Akka runs on the HotSpot JVM, it is optimized for long-running throughput by leveraging the Just-In-Time (JIT) compiler to perform aggressive optimizations, such as method inlining and escape analysis, on "hot" code paths. However, this performance comes at a cost of higher memory pressure as each Akka actor is a JVM object that requires significant heap allocation and contributes to the workload of the global Garbage Collector (GC), which can lead to "stop-the-world" pauses and increased energy draw during cleanup phases.

Regarding distribution, Akka achieves scalability through the *Akka Remote* and *Akka Cluster* modules. It provides "Location Transparency" by using *ActorPath* and *ActorRef* abstractions, allowing the system to send messages to remote nodes using the same semantics as local ones. However, spawning actors on remote nodes requires explicit configuration of serializer protocols and remote deployment descriptors, as distribution is treated as a high-level library feature rather than a language primitive.

2.2 Elixir on the Beam VM

Elixir is a modern functional language created by José Valim in 2011. While it provides a contemporary syntax and powerful tooling, it is built entirely on the Erlang Virtual Machine (BEAM), which has been refined for over 30 years to handle massive concurrency in the telecommunications industry [1]. In Elixir, the actor model is not a library but the core of the language itself.

Actors in Elixir are identified as "processes" that are significantly different from OS-level processes; they are lightweight, VM-managed entities that require as little as 2–3 KB of memory upon initialization. The BEAM architecture follows a "copy-nothing" principle. Unlike the JVM, where objects share a global heap, every Elixir process has its own private heap and its own dedicated

garbage collector. This is a critical factor for energy efficiency: when a process finishes its task and dies, its memory is reclaimed immediately without affecting the rest of the system or requiring a global GC pause. This total isolation prevents the memory-related energy spikes often observed in JVM-based systems during heavy orchestration.

Elixir's native support for distributed computing is one of its most distinguishing features. The BEAM was designed from the beginning to function as a single, distributed system. The ability to spawn an actor on a remote node is a primitive operation with the Node.spawn/2 function that allows a developer to create a process on any connected node as easily as on a local one. Remote communication is handled via a transparent distribution protocol that manages serialization and node-to-node heartbeats automatically.

2.3 Related work

In the context of our work two sets of related work are of principal importance: work on the energy consumption of actor-based systems and, more generally, work on the profiling of actor systems, in particular performance profiling. We present the most relevant references of these two corresponding bodies of work in the remainder of this section.

2.3.1 Actors and energy consumption. While research on the energy efficiency of distributed software has come into focus in recent years, few studies specifically address actor-based systems. Early means to bring energy awareness to the actor paradigm were proposed by Hayduk et al. [7], who demonstrated that the isolation properties of the actor model makes it an ideal foundation for managing power in heterogeneous architectures. By using Akka to offload specific tasks from CPUs to GPUs, they achieved energy savings of 40–80% compared to CPU-only executions. This work is further supported by the ParaDIME project [14], which proposed a full-stack infrastructure for data centers where Akka actors serve as the primary abstraction for energy-aware message passing and hardware-level power management.

More specific studies have investigated the energy characteristics of the runtimes themselves. Nagy et al. [11] developed the *GreenErl* framework to profile Erlang applications, finding that the choice of internal data structures and the use of the High-Performance Erlang (HiPE) compiler could reduce energy consumption by up to 50%. On a broader scale, Pereira et al. [12] conducted an extensive comparison of 27 programming languages, ranking them based on their energy efficiency across various benchmarks. Their results positioned Java as one of the most efficient languages, significantly outperforming Erlang in compute-intensive tasks. However, these rankings are primarily based on sequential workloads, which do not account for the massive concurrency and distribution costs found in modern actor systems.

2.3.2 Actors and execution performance. Apart from these energy-focused studies, a significant body of literature has evaluated actor frameworks based on traditional performance metrics like latency and throughput. Cardoso et al. [3], for instance, analyzed the performance of six actor and agent-based languages including Erlang and Akka using three concurrency benchmarks to evaluate reactivity and fairness. Their results indicated that the Erlang runtime provided the best overall performance for high-concurrency message passing, significantly outperforming Akka in terms of elapsed time for these specific orchestration tasks.

Similarly, Valkov et al. [19] analyzed Erlang, Go, and Scala with Akka in the context of server software engineering. Their evaluation of process creation time and communication throughput concluded that Erlang and Go excel at minimizing message latency and creation time. However, they noted that Akka was capable of supporting the largest number of "dormant" processes due to the JVM's efficient memory management for inactive objects.

Beyond performance, research has sought to understand the internal behavior of actor frameworks through instrumentation. Rosa et al. [16] developed a novel profiling tool for Akka based on the DiSL bytecode instrumentation framework to collect fine-grained metrics on creational overhead and executed work at the instruction level. Their results helped identify load-balancing issues in distributed frameworks like Apache Spark and Flink, proving that actor utilization is often uneven.

A recent study by Randtoul and Trinde [13] introduced the Supervised Communicating Processes (SCP) benchmark to evaluate the reliability of actor-based languages as Erlang and Scala with Akka. Their results demonstrated that Erlang maintained a 10% higher throughput than Akka in failure-free scenarios, whereas Akka exhibited greater performance deviations under high load, which they attributed to JVM garbage collection cycles and message latency fluctuations.

While these studies provide critical insights into the performance and reliability trade-offs of the two architectures, as well as the energy characteristics of their respective virtual machines, they do not investigate a direct comparative analysis of the energy efficiency of Akka and Elixir. Our research extends these findings by investigating how systematic architectural differences translate into physical energy consumption within a distributed environment.

Furthermore, most existing profiling research is restricted to single-node environments or relies on software-based power estimation. While tools like Intel RAPL [10] are frequently used to check energy consumption, they often fail to capture the total physical power draw of distributed components, such as network interface cards and memory controllers. Our work builds upon these performance studies by shifting the focus to high-resolution physical energy measurement in multi-node settings.

3 Experimental setup

In this section we present the hardware and software infrastructure that we have used to define and execute our benchmarks, the benchmark suite itself, as well as how we have automated benchmark deployment and executing using a specialized tool.

3.1 Hardware and software infrastructure

3.1.1 Grid'5000 infrastructure. All experiments were conducted on this Cloud infrastructure [2, 6], a highly reconfigurable scientific infrastructure for the experimentation of applications, algorithms, and infrastructures for parallel and geo-distributed computing.

Grid'5000 consists of approximately 500 compute nodes (15000 cores) organized in 40 clusters that are located in 12 different cities

in 3 countries (France, Belgium and Luxemburg). It is a major component of the SLICES RI [17] infrastructure, the European public research infrastructure for research on the Cloud and AI.

For this study, we used the Nova cluster at the Lyon site, which comprises 23 nodes, each being a Dell PowerEdge R430 server equipped with two sockets totaling 16 physical cores and 32 logical threads. To capture the physical ground truth of energy consumption, we utilized the cluster's integrated physical wattmeters measuring the actual electrical current and voltage at a resolution of 20ms. By conducting both single-node and multi-node experiments on identical hardware, we eliminated performance variables related to CPU architecture or memory bandwidth, focusing strictly on the energy characteristics of the virtual machines.

3.1.2 Software infrastructure. Grid'5000, as a research infrastructure, provides a complete set of tools for the automated reservation, deployment and execution of software artifacts, such as Python functions, containers or virtual machines.

We have developed a specialized tool in form of an easily configurable Python script for the automated deployment and execution of energy benchmarks on Grid'5000. Concretely, our tool consists of a Python script with easily configurable benchmark deployment and execution. It is based on EnosLib [4, 18], a library for "Infrastructure-as-Code" (IaC) management and automation of distributed experiments on Grid'5000. The central component of our tool is the interaction with the OAR Scheduler, the primary resource manager of Grid'5000, which manages, in particular, reservations of hosts, CPUs, GPUs, hard disk space and network resources. This tool and the automated benchmark handling is described in detail below, see Sec. 3.3.

3.2 Benchmarks

The benchmarks used in this study were adapted from the Savina benchmark suite [9], a standardized collection of workloads designed to evaluate the performance and scalability of actor-based systems. To obtain a comprehensive energy profile, we selected one benchmark to establish a baseline, and three benchmarks that are representative for actor usage.

(1) **Idle State:** This benchmark measures the passive energy consumption required to maintain 10^5 spawned actors; measurement begins only after a synchronization phase to ensure all actors have finished initialization. This allows us to observe the baseline cost of maintaining the runtime environment and identify the energy required just to keep the actor system alive.
(2) **Actor Spawning:** This benchmark measures the energy consumption of the principal operations of actor lifecycle management. It repeatedly creates and terminates large numbers of actors, thereby isolating runtime costs related to memory allocation, process initialization, scheduling, and registration within the virtual machine.
(3) **Message Passing:** This benchmark evaluates communication efficiency by repeatedly exchanging a token between two actors for a fixed number of iterations. The workload isolates message-passing overheads, including mailbox operations, message handling, and latency introduced by the underlying communication subsystem.

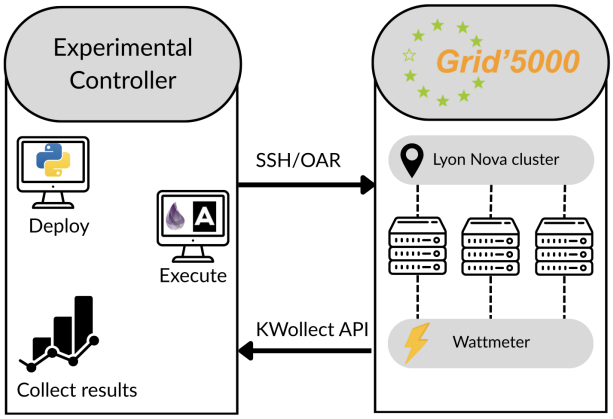


Figure 1: Experimental Architecture

(4) **Trapezoidal Approximation:** This benchmark assesses the energy consumption under CPU-intensive workloads by computing an approximation of an integral expression using a master–worker architecture. It performs extensive floating-point computations, principally trigonometric functions. In particular, this compute-intensive benchmark enables comparing the impact of the JVM's Just-In-Time (JIT) compilation with the execution model of the BEAM virtual machine.

3.3 Automated benchmark deployment and execution

In order to facilitate the execution of our energy benchmarks, in particular within the distributed Grid'5000 infrastructure, to achieve a high degree of reproducibility as well as to eliminate the errors associated with manual benchmark deployment and execution, we have developed a specialized tool for the automated deployment and execution of benchmarks for the Akka and Elixir actor systems.

We have based this tool on *EnosLib* [4, 18] and the OAR Scheduler, the primary resource manager for Grid'5000. OAR is responsible for managing job submissions, enforcing wall-time limits, and ensuring exclusive access to physical nodes. By interfacing with OAR via Python, our pipeline automatically reserved homogeneous nodes, ensuring that every benchmark was executed on an empty machine where no other user processes could interfere with the physical power draw.

Figure 1 shows how the benchmarks are executed: an experimental controller that runs on a local machine outside of Grid'5000 deploys the benchmarking code on one or several nodes of the Nova cluster of Grid'5000, executes them there, reads the energy information from the wattmeters of the nodes and collects/returns them for analysis.

3.3.1 Single-Node benchmarking. We have first established a local baseline on a single Nova node. This required a execution protocol to ensure that the physical energy readings were not skewed by the system's previous state:

(1) **Interleaved Framework Execution.** Rather than running all Akka benchmarks followed by all Elixir benchmarks, the pipeline interleaved them. This eliminated systematic errors that affect both frameworks because of transient environmental factors, such as network jitter or ambient data center temperature shifts.
(2) **Baseline Recovery.** Between every individual run, the script enforced a 60-second cooling down period. This "sleep" period is essential in physical benchmarking to allow the CPU's active cooling fans to return to their idle RPM and to ensure the CPU temperature dropped to its baseline. Without this, a compute-heavy Akka run could cause a subsequent Elixir run to consume more energy simply due to residual thermal overhead.
(3) **Energy Retrieval and Post-Processing.** Our pipeline handled data extraction automatically. After each run, the Python orchestrator parsed the standard output of the virtual machines to identify the `LOG_START` and `LOG_END` Unix timestamps. These were then harnessed in the context of queries using the **Kwollect API**. The script retrieved the raw wattage values for the specific node and time window, performing a numerical integration using the trapezoidal rule to convert instantaneous power (Watts) into total energy (Joules). This is defined by the following (approximage) equation:

$$E = \int_{t_{start}}^{t_{end}} P(t)\,dt \approx \sum_{i=1}^{n-1} \left(\frac{P(t_i) + P(t_{i+1})}{2} \right) \Delta t_i \qquad (1)$$

where E is the total energy in Joules, $P(t)$ represents the instantaneous power draw in Watts at time t, and Δt_i is the sampling interval (nominally 1 second in our configuration).

3.3.2 Distributed benchmarking. Benchmarking of multi-node distributed configurations introduced significant technical challenges, particularly regarding framework configuration and network orchestration. In this setting, the automation script was expanded to manage a cluster of three nodes, where roles were dynamically assigned as *Master* (orchestrator) and *Worker* nodes.

(1) **Dynamic Role and IP Management.** Unlike local tests, distributed actor frameworks require explicit knowledge of the cluster's network topology. Our EnosLib script automated this by fetching the internal IP addresses of the OAR-assigned nodes. For Elixir, these IP addresses were used to configure the `-name` and `-cookie` parameters of the BEAM VM, allowing for near-instantaneous cluster formation.
(2) **Challenges of the Akka distribution model.** The implementation of distributed Akka proved significantly more complex than Elixir. While Elixir provides native, language-level primitives for remote spawning (`Node.spawn`), Akka treats distribution as a library-level feature. This required our automation script to manage several layers of configuration using the `akka.remote` package. Akka requires explicit serializer configuration to handle the transformation of Java objects into bytes for network transmission.
(3) **The Provisioning Phase.** To ensure a scientific "ground truth," our script began every distributed experiment by provisioning the nodes from scratch. Each node was flashed with a clean Ubuntu 22.04 image, followed by the automated installation of the OpenJDK and Erlang runtimes. This approach guarantees that the results are not biased by configuration or leftover files from previous experiments.

By automating this entire lifecycle from the VM installation to the final integrated Joule calculation, we provide a benchmarking framework that is both transparent and highly resilient to the complexities of distributed infrastructure.

4 Discussion

We have examined the performance of Akka and Elixir across four distinct scenarios, evaluating both single-node and distributed configurations.

In the remainder of this section, we present the results for the four benchmarks introduced above. All data points represent values of energy consumption in Joules and duration in seconds, averaged over three independent runs of each benchmark.

Table 1: Energy consumption and duration of single-node benchmarks

Benchmark	Language	Avg. Joules	Avg. Duration (s)
Idle	Akka	1934.72	30.00
	Elixir	1877.39	30.00
Message	Akka	2383.91	30.67
	Elixir	609.52	7.00
Spawn	Akka	1257.65	13.00
	Elixir	1048.20	13.00
Trapezoid	Akka	1192.34	9.67
	Elixir	4888.24	36.67

4.1 Idle State

As illustrated in the single-node results (Table 1), the energy consumption for a 30-second window is remarkably similar between the two frameworks, with Akka consuming 1,934.72 J and Elixir consuming 1,877.39 J. Given this similar pattern in the single-node configuration (Figure 2), an idle benchmark for the distributed configuration was not included in the analysis, as the idle draw is the linear aggregate of the individual nodes' baseline consumption.

The minor difference observed suggests that both frameworks have a very similar baseline energy footprint when the system is at rest. This finding is significant as it proves that any substantial energy divergence observed in subsequent benchmarks is a direct result of how the frameworks manage orchestration or computation, rather than an inherent difference in the power draw of the virtual machines. It further validates our experimental environment, confirming that the background noise from the operating system and the hardware on the Nova cluster nodes is consistent across both test groups.

4.2 Actor Spawning

The Spawn benchmark evaluates the energy cost associated with the actor lifecycle, specifically the overhead of creating and destroying large numbers of actors. This scenario isolates the performance

Table 2: Mean Distributed Energy Metrics Across Three Iterations

Benchmark	Language	Avg. Joules	Avg. Duration (s)
Message	Akka	129,102.95	829.24
	Elixir	15,473.05	99.50
Spawn	Akka	15,431.59	98.08
	Elixir	1,801.36	11.01
Trapezoid	Akka	36,951.48	161.99
	Elixir	71,534.83	302.69

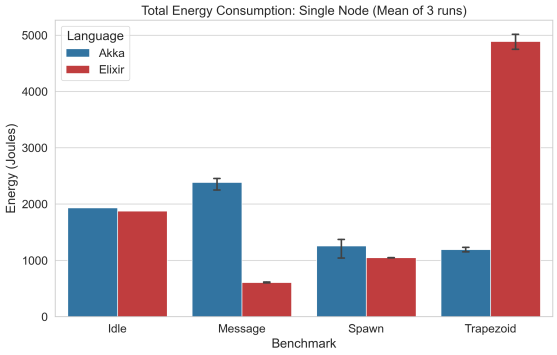


Figure 2: Total energy consumption (Joules) for Akka and Elixir in the single-node configuration.

of memory allocation and the efficiency of the virtual machine's process management.

In the single-node configuration, the energy consumption between the two frameworks is relatively close: Akka consumed an average of 1, 257.65 J while Elixir consumed 1, 048.20 J. In the distributed configuration, however, Elixir's energy consumption remains remarkably low at 1, 801.36 J, while Akka's consumption increases to 15, 431.59 J, a nearly 8.5x difference (see Table 2). As illustrated in the single-node power profile (Fig. 3), even though the curves for the Spawn benchmark follow the same trajectory for both frameworks, the Elixir curve is positioned significantly lower on the graph than the Akka curve, thus confirming that Elixir consumes less power throughout the entire duration of the actor spawning task.

In Elixir, processes are a native primitive of the BEAM virtual machine, requiring only approximately 2 KB of memory and utilizing private heaps that are garbage-collected independently from one another. In contrast, Akka actors are implemented as objects on top of the JVM, so creating 10^5 actors in a distributed environment requires significant memory allocation and triggers the JVM's Garbage Collector, which must manage a much larger object graph. Furthermore, Elixir's native support for remote spawning allows it to initialize actors on worker nodes with minimal network orchestration, whereas Akka has the additional runtime and communication overhead caused by the Akka Remote library that manages remote actor deployment and lifecycle synchronization.

4.3 Message Passing

The Message benchmark measures the energy efficiency of asynchronous communication by repeatedly exchanging messages between nodes. On a single node, Elixir demonstrates a clear efficiency advantage, consuming 609.52 J compared to Akka's 2, 383.91 J. In this local setting, Elixir is four times more efficient and significantly faster, completing the task in 7 s versus Akka's average of 31 s. When moved to the distributed configuration, this gap scales in magnitude: Elixir consumes 15, 473.05 J, while Akka consumes 129, 102.95 J (Fig. 5), representing an 8.3x energy efficiency gap.

As seen in the distributed experiments data, Akka takes nearly 14 minutes (829.24 s) to complete the same messaging workload that Elixir finishes in just over 1.5 minutes (99.50 s) (see Table 2). Because Akka is slower at processing the message queue and handling remote serialization, the hardware remains in an active, high-power state for over eight times longer than the Elixir. The impact of the physical network boundary is most visible in the distributed power profile (see Fig. 4). Unlike the relatively smooth power draw observed in the single-node Message benchmark (Fig. 3), the distributed case exhibits significant fluctuations and 'noise', which is particularly pronounced in the Akka implementation.

The power profiles confirm that the "energy tax" for Akka is largely a product of time; while the instantaneous power draw is comparable between the two, Akka's inability to process the messaging backlog quickly forces the system to integrate the per-message power draw over a much longer time window. This suggests that for communication-heavy, IO-bound distributed systems, Elixir's low-latency architecture provides a significantly more sustainable foundation than the JVM's object-passing model.

4.4 Trapezoidal Approximation

The results for the Trapezoidal benchmark evaluates pure numeric performance during a CPU-intensive task. It reveals a significantly different picture than the orchestration-heavy benchmarks.

As shown in Table 1, for the single-node configuration, Akka is nearly 4x more energy-efficient than Elixir, consuming only 1, 192.34 J compared to Elixir's 4, 888.24 J. In the distributed configuration, Akka maintains a lesser advantage, consuming 36, 951.48 J while Elixir consumes 71, 534.83 J, see Fig. 5. In both cases, Akka completes the computation significantly faster than Elixir.

The power profiles in Fig. 4 provide the scientific explanation for this performance. Akka exhibits a sharp initial power spike, which represents the energy consumed by the JVM's JIT compiler as it analyzes and optimizes the code paths. However, once the math-heavy code is compiled into native machine instructions, Akka performs the numeric integration task with high efficiency. Elixir (on the BEAM VM), being designed for pattern matching and low-latency communication rather than floating-point math, lacks this level of numeric optimization, resulting in a constant, higher power draw that lasts nearly twice as long as Akka's execution.

4.5 Evaluation of Ahead-of-Time (AOT) Compilation

To further validate this result, we conducted another experiment by disabling the JIT compiler using the `-Xint` flag. In this configuration, Akka's energy efficiency dropped dramatically, with the total

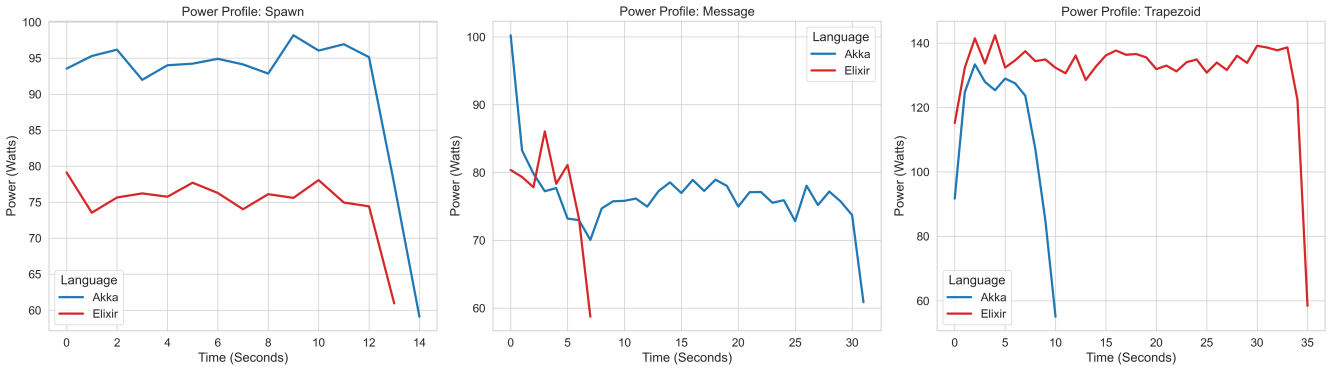


Figure 3: Single Node Power Profiles in Watts

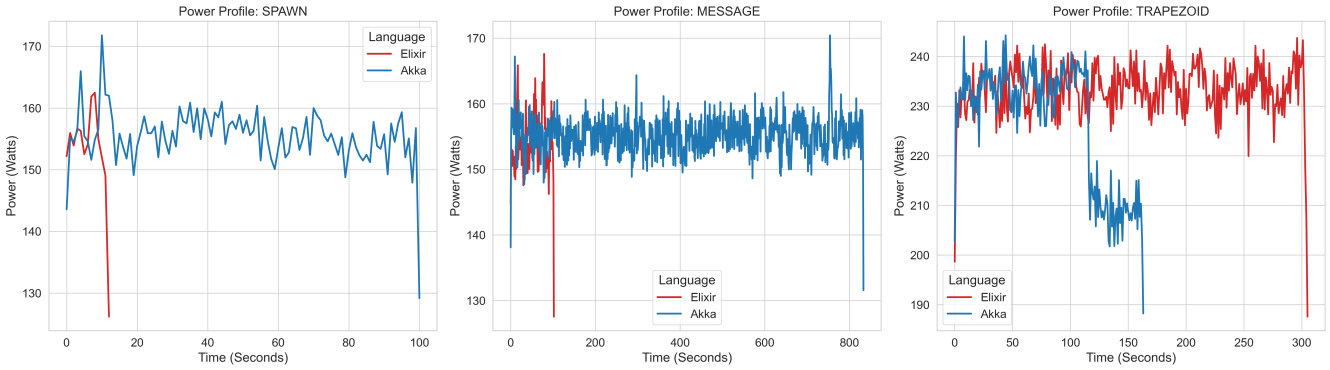


Figure 4: Distributed nodes Power Profiles in Watts

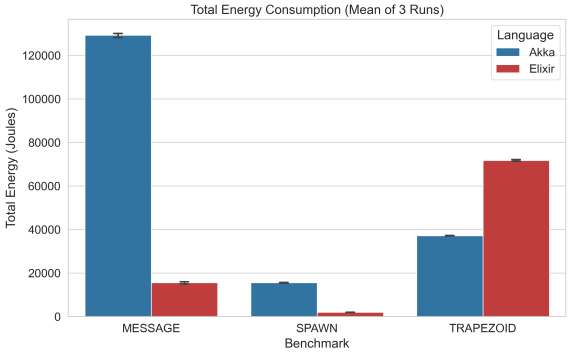


Figure 5: Aggregate nodes energy consumption (Joules) in the distributed configuration

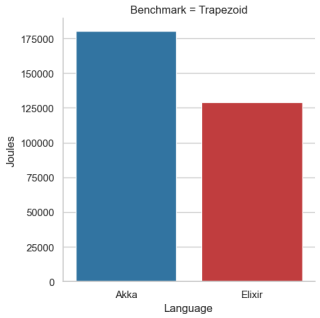


Figure 6: Trapezoidal benchmark with disabled JIT

energy consumption exceeding those of Elixir (see Fig. 6). This result confirms that Akka's computational superiority is not inherent to the actor model itself, but is a direct benefit of the JVM's optimization engine, which "amortizes" its initial energy investment during long-running CPU-intensive tasks.

Beyond standard JIT execution, we attempted to evaluate the energy efficiency of Akka using GraalVM Native Image to reduce the energy consumed during initial startup and the high memory footprint associated with the traditional Java Virtual Machine.

However, using AOT compilation for standard distributed Akka applications remains a significant technical challenge. GraalVM requires all reachable code paths to be statically identified at compile time. In contrast, standard Akka relies heavily on dynamic

features of the JVM, including Java Reflection for actor instantiation, dynamic class loading for resolving actor hierarchies, and runtime configuration. Because these operations are resolved only at runtime, the AOT compiler cannot statically guarantee their safety, leading to terminal build failures and runtime initialization crashes. While recent industry developments such as *Akka Edge* have introduced restricted subsets of the framework designed for AOT compatibility, the standard Akka modules required for the Savina benchmarks are effectively tethered to the JIT-compiled environment.

5 Conclusion and future work

This study provides comparative energy benchmarks of the Akka and Elixir actor frameworks using physical wattmeters on the Grid'5000 infrastructure. Our findings demonstrate that the most sustainable choice depends entirely on the nature of the workloads executed using actors.

Our empirical results establish a clear trade-off between orchestration and computation: Elixir is the much better choice for orchestration-heavy, IO-intensive applications, achieving an up to 8.5x energy reduction for actor spawning and an 8.3x reduction for message passing. This is attributed to the BEAM VM's native, lightweight process model and private heap architecture. Conversely, Akka is nearly 2x more efficient for CPU-bound numeric computation, leveraging the JVM's JIT compiler to optimize math-intensive code into native instructions.

For software architects, these results suggest that Elixir is the most sustainable foundation for high-concurrency "routing" or "messaging" systems, such as instant messaging backends or IoT gateways. Akka, on the other hand, remains the preferred choice for distributed systems requiring heavy data processing and complex calculations.

In future work, we plan to leverage these findings to develop Energy Contracts within the Elixir actor framework. These contracts provide a formal mechanism for developers to define energy-aware policies, allowing distributed applications to dynamically adapt their execution and orchestration based on real-time power availability and physical constraints.

Acknowledgments

This project has received co-funding from the European Union's Horizon Europe Research and Innovation Programme under Grant Agreement No. 101126644. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Joe Armstrong. 2007. A history of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) *(HOPL III)*. Association for Computing Machinery, New York, NY, USA, 6–1–6–26. doi:10.1145/1238844.1238850

[2] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. 2006. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications* 20, 4 (2006), 481–494.

[3] Rafael C. Cardoso, Jomi F. Hübner, and Rafael H. Bordini. 2013. Benchmarking Communication in Actor- and Agent-Based Languages. In *Engineering Multi-Agent Systems*, Massimo Cossentino, Amal El Fallah Seghrouchni, and Michael Winikoff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 58–77.

[4] Ronan-Alexandre Cherrueau, Marie Delavergne, Alexandre van Kempen, Adrien Lebre, Dimitri Pertin, Javier Rojas Balderrama, Anthony Simonet, and Matthieu Simonin. 2022. EnosLib: A Library for Experiment-Driven Research in Distributed Computing. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (2022), 1464–1477. doi:10.1109/TPDS.2021.3111159

[5] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Amsterdam, Netherlands) *(AGERE 2016)*. Association for Computing Machinery, New York, NY, USA, 31–40. doi:10.1145/3001886.3001890

[6] Grid'5000 consortium. [n. d.]. Grid5000 home. https://www.grid5000.fr. last accessed: 25 July 2026.

[7] Yaroslav Hayduk, Anita Sobe, and Pascal Felber. 2016. Enhanced energy efficiency with the actor model on heterogeneous architectures. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, Cham, Switzerland, 1–15.

[8] Carl Hewitt. 2010. Actor Model of Computation: Scalable Robust Information Systems. arXiv preprint arXiv:1008.1459. Version 2. https://arxiv.org/abs/1008.1459

[9] Shams M. Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control* (Portland, Oregon, USA) *(AGERE! '14)*. Association for Computing Machinery, New York, NY, USA, 67–80. doi:10.1145/2687357.2687368

[10] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. 2018. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 2 (2018), 1–26.

[11] Gergely Nagy, Áron Attila Mészáros, István Bozó, and Melinda Tóth. 2019. Tools supporting green computing in Erlang. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang* (Berlin, Germany) *(Erlang 2019)*. Association for Computing Machinery, New York, NY, USA, 30–35. doi:10.1145/3331542.3342570

[12] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (Vancouver, BC, Canada) *(SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 256–267. doi:10.1145/3136014.3136031

[13] Aidan Randtoul and Phil Trinder. 2022. A reliability benchmark for actor-based server languages. In *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang* (Ljubljana, Slovenia) *(Erlang 2022)*. Association for Computing Machinery, New York, NY, USA, 21–32. doi:10.1145/3546186.3549928

[14] Santhosh Kumar Rethinagiri, Oscar Palomar, Anita Sobe, Gulay Yalcin, Thomas Knauth, Rubén Titos Gil, Pablo Prieto, Malte Schneegaß, Adrian Cristal, Osman Unsal, et al. 2015. ParaDIME: Parallel distributed infrastructure for minimization of energy for data centers. *Microprocessors and microsystems* 39, 8 (2015), 1174–1189.

[15] Raymond Roestenburg, Rob Williams, and Robertus Bakker. 2016. *Akka in Action*. Manning Publications, Shelter Island, NY, USA.

[16] Andrea Rosà, Lydia Y. Chen, and Walter Binder. 2016. Profiling actor utilization and communication in Akka. In *Proceedings of the 15th International Workshop on Erlang* (Nara, Japan) *(Erlang 2016)*. Association for Computing Machinery, New York, NY, USA, 24–32. doi:10.1145/2975969.2975972

[17] The SLICES-RI consortium. [n. d.]. SLICES-RI. https://www.slices-ri.eu/. last accessed: 25 July 2026.

[18] The STACK team. [n. d.]. EnOSlib. https://gitlab.inria.fr/discovery/enoslib. last accessed: 25 July 2026.

[19] Ivan Valkov, Natalia Chechina, and Phil Trinder. 2018. Comparing languages for engineering server software: erlang, go, and scala with akka. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 218–225. doi:10.1145/3167132.3167144