Lesser Evil: Embracing Failure to Protect Overall System Availability

Viktória Fördős[1,2][0000−0001−6403−9797] and
Alexandre Jorge Barbosa Rodrigues[1][0000−0001−7618−6743]

[1] Cisco Systems, Stockholm, Sweden
[2] ELTE, Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary
{vfordos,albarbos}@cisco.com

Abstract. Low memory conditions degrade system performance, challenge programs to fulfil their SLA and can lead to out-of-memory errors causing a major system outage. Running low on memory is an especially dangerous situation in case of mission critical, embedded Erlang systems with high availability requirements. In such systems, total system outage must be avoided at all costs. Nonetheless, no solution exists today that can be added to an Erlang system without code modification and would treat memory pressure out of the box.
We propose an approach, called Lesser Evil, that can treat low memory pressure in any fault-tolerant Erlang system without the need of any code modification. Our experiments suggest that, with the help of Lesser Evil, an embedded Erlang system can survive low memory conditions and avoid a major outage.

Keywords: Embedded systems, Memory management, Fault tolerance, Erlang.

1 Introduction

In our digitalised world we depend on mission critical, embedded systems. These systems need to be always available regardless of the current load on the system. Parallelism is a key enabler to accommodate a system to variable workload. However, what stays constant in a system is the total memory installed (per a single host). If a system runs low on available memory, it will not perform as expected. Since it is struggling to allocate memory enough to execute requests, it will not satisfy its QoS metrics, neither SLA requirements. Considering best case scenarios, its performance will degrade while in the worst case the program will stop with an out-of-memory error causing the system to reboot. Taking into account that embedded systems have strongly constrained resources and slow processing capabilities, the reboot can take significant amount of time while the system stays offline. Hence, running low on memory is a critical event in case of mission critical, embedded systems that should not be ignored as it can compromise the availability of the system.

The presented problem is especially relevant in case of embedded systems that are written in Erlang. The Erlang programming language was designed in the

Ericsson software technology lab for systems that will never stop or fail. Early use cases of Erlang at Ericsson include a multi-service switch, AXD 301 [12], where the control system was implemented in Erlang. Since then, Erlang has become well-known and used in the telecommunication segment, multiple companies have decided to adapt Erlang to implement mission critical, embedded systems. An illustrative example of the significance of Erlang in this sector is that Cisco ships about 2 million devices per year with Erlang in them and that 90% of all internet traffic goes through Erlang controlled nodes [4].

Motivation. Running low on memory is an especially dangerous situation in case of mission critical, real-time embedded systems with high availability requirements. In such systems, total system outage must be avoided at all cost, even by allowing temporary, partial failures. Nonetheless, no solution exists today that can be added to an Erlang system without code modification and would treat memory pressure out of the box. Having such a solution would enable existing Erlang systems to perform better under low memory conditions. It would help any Erlang system to deal with unexpectedly large workload but *would be very beneficial for embedded systems* where virtual memory is disabled and physical memory is very limited and constrained.

Our contributions. In this paper, we propose an approach, called Lesser Evil, that can treat memory pressure in any Erlang system without the need of any code modification. Lesser Evil is ready to use and is applicable to any fault-tolerant Erlang systems. We discuss why our approach is viable, how it identifies low memory conditions, what strategy it employs and how it treats memory pressure.

Summary of results. Our evaluation shows that an Erlang system can survive and keep functioning under low memory conditions with the help of our approach. We have confirmed that Lesser Evil's strategy is correct: it is able to identify and execute compensating actions on the components most responsible for the situation without damaging the other components or the overall system.

2 Problem Statement

Let us take an imaginary, fault-tolerant, embedded Erlang system as a motivating example. The Erlang system is designed to run on and control an embedded device with limited memory and slow processing capability. The Erlang system and the device are expected to be always available. The Erlang system receives administrative requests that it processes, and as a result of processing, it executes commands on the embedded device. The Erlang system usually receives a few, small requests that never fills up the device memory, but sometimes a large request arrives that may require all the available memory of the device to process. If part of the device memory is used to process other small requests, the available memory is not enough to process the large requests: the device runs out of memory, becomes unresponsive or it even reboots.

To protect the availability of the device, the only available solution to use today is a generic solution for Linux systems, called the Out of Memory Man-

ager (OOM manager) [23]. However, the OOM manager is not an option in our example, as the OOM manager would terminate the whole Erlang system that controls the device; making the device unavailable. A more fine grained solution would be more appreciated that would not cause the termination of the whole Erlang system. Instead, it would free up enough memory to resume the normal operation of the Erlang system, for example, by terminating the large request.

The motivating example illustrates that we need a solution that treats memory pressure differently. The primary goal is to free up enough memory to resume the normal operation of the Erlang system. When releasing memory we accept local failures but we cannot tolerate abnormal termination of the whole Erlang system neither the risk of introducing permanent data inconsistencies. Under local failures we mean failures that may effect some end-user but not all users and also failures that may result in interruption of some requests but not all requests. Our goal is to prevent a major outage, but we accept temporary, partial system degradations.

To achieve our goals, a tool is needed that is able to monitor, assess and interact with the running Erlang system. The tool needs to monitor the running Erlang system to identify low memory conditions. If low memory conditions are identified the tool needs to select some entities to execute compensating actions. The tool needs to have a strategy to select entities and the strategy should consider multiple criteria when making the selection. The strategy should be based on metrics of the entities gathered from the runtime, and characterise the *badness* of the entity. Compensating actions need to be defined on entity level, ensuring that the impact of the action is limited; will not have a negative impact on the overall Erlang system.

3 Erlang

In this section we discuss the Erlang ecosystem and highlight its key features that our approach uses to treat memory pressure.

Erlang [9] is a dynamically typed functional programming language that has built-in support for concurrency and distribution. Erlang systems are well known about their high availability, thanks to their failure handling and hot code loading capabilities.

An Erlang system can be considered as an actor based system, built up of interacting Erlang processes. Erlang processes [14] are light-weight processes with small memory overhead, fast to create and terminate, and the scheduling overhead is low. The Erlang processes communicate with each other via asynchronous message passings [19]. They share nothing with the outside world, they exclusively own their data (excl. large binaries that are reference counted and stored in a shared heap). If data is shared between two processes (via message passing), the data is copied to the another process's heap. Processes, while scheduled, are executing code, which is measured in *reductions*. The scheduler [20] de-schedules a process after a certain amount of reductions or when the process enters into a waiting state (e.g. waiting for messages or IO). The fact that processes share

nothing and that they can yield to be de-scheduled enables Erlang's garbage collector [15] to work on a single process at a time without interfering with other running processes. Moreover, when an Erlang process terminates all its allocated memory can be freed as there are no other users to those memory blocks.

Fault tolerance is built into the language [13], processes are allowed to fail. Hence, interacting processes can never assume that their peer is alive. To detect the termination of their peer, processes can monitor other processes. If the monitored process terminates, the process will be notified by a message. If the interacting processes depend on each other they can use another language construct, called the process link. A process link is a bidirectional link forwarding events of termination, called error signals, between the linked processes. The forwarded error signal may terminate the receiving process as well, depending whether the process trapping exits. Using links and monitors more complex and robust architectures can be built that define restart strategies for the important processes. An example is the supervisor process [18] that is responsible for starting, terminating and restarting its children. A common practice is to build supervisor trees that allow managing processes in a structured way and handling runtime errors where it is convenient.

From the system architecture point of view, an Erlang system is built up from a single or a set of Erlang nodes organised into a full mesh cluster. An Erlang node, which is an OS process, is built up from the Erlang Runtime System (ERTS) and the Erlang release. An Erlang release [17] consists of a set of Erlang applications. The Erlang applications [16] are the reusable units of the Erlang ecosystem, can be bundled into multiple releases and be part of different deployments. (Readers experienced in the Java programming language can consider Erlang applications as Java packages.) An Erlang application is the implementation of a functionality and can cooperate with other Erlang applications to implement more complex functionality. An Erlang application defines and includes its Erlang processes. When an application starts the runtime starts an application_master process as the main responsible process for the application. The application_master in turn starts a process called 'x' that is responsible for the IO and to start the main supervisor for the application. The main supervisor is the first processes implemented by the user, the root of the supervisor tree for the application. The ERTS enables interaction with the running Erlang node. Processes can be created and terminated, and existing processes can be inspected at runtime: various process metrics can be retrieved. Applications can be started, stopped and upgraded without stopping the Erlang node.

4 Lesser Evil

In this section we describe our approach for Erlang systems, called Lesser Evil [22]. First, we outline the approach and then we discuss the details: what entities are considered, how badness is characterised, how the strategy is evaluated and what compensating actions are employed. We show the architecture and discuss the different ways the approach can be added to an Erlang system.

Our approach aims at treating memory pressure of an Erlang system without the need of code modification. It proposes to monitor the running program and upon low memory conditions select some entities with the greatest badness values and execute compensating actions on the selected entities. Mapping the approach to Erlang systems, the approach monitors an Erlang node and collect its memory consumption. As Erlang processes are owning their data, they hold memory in the system, thus the Erlang processes are the entities. The goal is to characterise the badness of processes with the help of process level metrics. As the runtime provides interaction possibilities with the running system, the approach employs two compensating actions: triggering garbage collection on a selected process, and terminating the process.

4.1 Entities

Lesser Evil considers Erlang processes as entities. However, it does not consider all processes, as there are critical processes in an Erlang node. As examples, consider the various system processes started by the runtime and the processes belonging to critical user applications. Automatically identifying and excluding the system processes is possible, however, distinguishing between critical and non-critical user processes is not. Lesser Evil takes the approach of letting the user decide, and requires a list of Erlang applications that are non-critical for the system (e.g. the implementation of an HTTP API). Processes belonging to the listed applications are considered only. However, this is still not enough. As discussed in Section 3, each application includes 3 critical processes (application_master, process 'x' and the main supervisor of the application) that must be excluded. If they terminate, the whole application terminates, which must be prevented.

4.2 Badness

Determining the badness of processes is the heart of the strategy: the compensating actions executed on the selected processes should help the Erlang node to survive low memory conditions, and their negative impact on the rest of the processes should be minimal. In this section we discuss how such a metric can be constructed.

To treat memory pressure effectively, the badness assigns greater values to processes that (1) have high memory usage; (2) have several pending tasks, therefore, they have a bad future outlook.

To minimise the negative impact of the compensating actions on the rest of the processes, the badness assigns lower values to processes that (1) are long-lived and, therefore, have proven their good behaviour; (2) are important to the user; (3) play a central role in the system in the sense of several processes depend on them.

In conclusion, the goal is to select isolated, relatively new processes with high memory usage and a bad future outlook. Therefore, the *badness* is a composite

metric that assigns a real value to a process based on the following process level metrics.

- *Memory*. Number of bytes the process uses.
- *MessageQLength*. Number of messages delivered to but not yet processed by the process. The metric expresses how much more tasks a process has pending.
- *Reds*. Reduction count shows the amount of work the process has done. As for long lived processes this value can be several order of magnitudes large, the logarithm with base 10 is applied to the reduction count in the badness formula.
- *Age*. Number of checks the process has stayed alive.
- *Links*. The number of processes linking to the process. The metric expresses how central the process is.
- *Mons*. The number of processes monitoring the process. The metric expresses how central the process is.
- *Prio*. Erlang processes can have different priorities. The higher the process priority is, the more important the process to the user is. This is what the metric expresses by selecting a value from $\{1, 10, 100\}$.

The *badness* metric that assigns a real value to a process can be formalised as shown by

$$badness \equiv \frac{Memory * (MessageQLength + 1)}{log_{10}(Reds) * (Links + 1) * (Mons + 1) * Age * Prio}$$

4.3 Strategy

In this section we define the strategy and how it is evaluated. The strategy works with an Erlang node, and takes the memory limit, further denoted as *MemLimit*, the Erlang node can occupy as a configuration parameter.

The strategy maintains a state to store historical data about the processes (i.e. age) and about the compensating actions executed in the past. Data about the past actions is necessary to prevent cascading failures that would be caused by too frequent compensating actions. Thus, the strategy ensures a cool down interval is respected between two compensating actions.

The strategy is invoked when new system and process level metrics arrive. The strategy decides whether compensating actions are required, executes actions if required and updates its state. To decide whether compensating actions are required, it first checks if it is not in cool down interval (*NotInCoolDownInterval*) by testing that a certain amount of seconds (currently, it is 5 seconds) has elapsed since the last action was executed. We have chosen to use 5 seconds as cool down interval to minimise the risk of cascading process failures. Then, it looks at the system level metrics, namely, memory used by the Erlang node (*Mem*) to check whether the current memory allocations are close to the maximum. To summarise, the strategy triggers as follows.

$$trigger \equiv Mem > 0.8 * MemLimit \ \wedge NotInCoolDownInterval$$

When compensating actions are required the strategy selects a compensating action as follows.

$$select\_action \equiv \begin{cases} trigger\_gc & \text{if } Mem < MemLimit \\ terminate\_proc & \text{otherwise} \end{cases}$$

After that, the strategy orders the processes by their badness score that is calculated using the received process level metrics and historical data stored in its state. Now the goal is to counterbalance the low memory conditions by freeing up memory. Thus, it takes as many processes from the beginning of the process list as it needs to have the memory condition settled. The resulting set contains processes with the highest badness score.

4.4 Compensating Actions

The strategy has selected a compensating action and a list of processes to execute the compensating action on. We defined two compensating actions *trigger_gc* and *terminate_proc*. The *trigger_gc* action is to trigger garbage collection on the selected processes, while the *terminate_proc* action is to terminate the selected processes. The *trigger_gc* action is based on calling `erlang:garbage_collect/1` function in order to trigger a full sweep garbage collection of the process. The *terminate_proc* action is non-trivial, we discuss its details now. The action first sends a trappable exit signal to the process. The exit signal will terminate the process if it is not trapping exits, however, if the process is trapping exits, the exit signal is delivered as a message in its mailbox and it is up to the process to decide whether to terminate. Hence, the action waits a few milliseconds and checks if the process is still alive. If it is alive, the action sends a non-trappable exit signal that will terminate the process immediately. In order to avoid cascading process failures, the action waits 3 seconds between the termination of two processes.

4.5 Architecture

In this section we put things together: we show the main components of Lesser Evil and discuss how it can be applied to Erlang systems with different requirements.

Lesser Evil is organised into two Erlang applications: `lesser_evil` and `lesser_evil_agent`. The main responsibility of the `lesser_evil` application is to monitor the Erlang node(s) and to evaluate the strategy based on the metrics it receives from the agent(s). The application was designed in a way that it supports supervising multiple Erlang nodes. The `lesser_evil_agent` application is responsible for collecting and forwarding system and process level metrics to the `lesser_evil` application and executing actions it receives from the `lesser_evil` application.

In the rest of the section, we provide guidance on how to use Lesser Evil: how to deploy, configure and install Lesser Evil for Erlang systems with different requirements.

In case of deployments where the network is not reliable or not secure, the two applications should be included into the monitored Erlang node. Otherwise, the agent should be included in the monitored Erlang node but the `lesser_evil` application should be deployed as a standalone Erlang node. If the deployment has multiple nodes and the network is stable, secure and the bandwidth is not constrained, a central `lesser_evil` node handling all the nodes seems to be a better choice.

As the next discussion point we provide guidance on how Lesser Evil can be configured and added to an Erlang system. First of all, programmers need to choose the Erlang applications to be monitored by the agent code. The selected applications should be fault-tolerant applications that are non-critical from the system point of view. Adding the core parts of the persistence layer is advised against, while adding the northbound API, the cache layers, data consumers are encouraged. After selecting the entities Lesser Evil will work with, one need to decide on the memory limit that will be enforced by Lesser Evil. One should work with historical data, and aim to choose a memory limit that does not activate Lesser Evil during normal load and trigger garbage collection only for a bit more intense load scenarios. We recommend to perform load tests to choose the memory limit best fitting the Erlang system, and also to ensure that the applications Lesser Evil interacts with are prepared for failures.

4.6 Discussion

In this section we discuss the design decisions we have made and the implications Lesser Evil can have on the monitored node.

Badness We start the discussion with the badness metric (Section 4.2), that is the heart of the strategy that Lesser Evil employs.

One observation to make is that the messages are handled uniformly, nonetheless, they are not and they can have different implication on the process in real-life. Processing one message may take only a few reductions while handling another message can lead to thousands of reductions. Nevertheless, we choose to consider all messages uniform as processes usually have a quasi-empty mailbox in Erlang systems. Moreover, processes with large message queues are considered a performance bottleneck in an Erlang system, hence we believe it is justified to prioritise selecting them.

Another observation one may make is that due to the age factor older processes are more protected, that can lead to the starvation of new processes. However, we argue that this is not a problem. First of all, when Lesser Evil is triggered the system is already at risk, there is no point in initiating more work. Second, long lived processes are long lived for a reason: they are central parts of the system with possibly lot of dependent processes and important responsibilities. Therefore, their protection is necessary to avoid cascading failures and to limit the impact of system degradations.

Compensating actions Another discussion point is the compensating actions (Section 4.4) that are triggering garbage collection on process level or terminating a selected process.

Triggering immediate garbage collection can have side-effect on the process, as the process must not be executing code while the garbage collection occurs. Therefore, *if* the process is scheduled when the garbage collection is triggered, the process is going to be de-scheduled while the garbage collection takes place, resulting in longer execution times.

Terminating a process is a more serious event in the system. As an example, what happens with files opened by the terminated process? In Erlang, files are opened through an auxiliary process that is linked to the process that opened the file, implying that upon the termination of the process, the auxiliary process gets notified and will close the file. The same holds for other shared resources.

Another point to consider is the question of cascading failures. When a process terminates all processes that have monitors on or are linked to the terminated process get notified and may decide to terminate themselves, leading to cascading failures. Cascading failures further reduce the memory and increase the impact of the compensating action, which is meant to be kept minimal. To avoid cascading failures the badness metric down-prioritises processes that are central.

Furthermore, if the process was supervised, the supervisor is notified and decides on whether to restart the process. Too frequent restarts make the supervisor to give up: it terminates all its supervised processes and itself. The 5 seconds of cool down interval and the waiting time of 3 seconds between two process terminations are there to avoid such scenarios. The constants were determined based on reviewing popular, open-source Erlang applications, however, the authors believe that there can be cases where the constants need to be changed, hence they will become configuration parameters of Lesser Evil.

4.7 Note on Applicability

In 2011, the Elixir language [32] was introduced that runs on the same virtual machine, called the BEAM, as Erlang. Elixir has become a success, it is in the 48^{th} place on the TIOBE index [33] published in May 2021 [1]. Elixir has managed to attract even more companies, thus the BEAM has become more widespread. Processes started in Elixir have the same capabilities and properties as Erlang processes, thus Lesser Evil is able to monitor Elixir processes as well. Furthermore, as Erlang system level API functions used by Lesser Evil are available in Elixir systems and Erlang applications can be part of Elixir releases, Lesser Evil works not only with Erlang systems but with Elixir systems as well. This fact greatly increases the applicability of our approach.

5 Evaluation

In this section we present the evaluation of Lesser Evil. The primary use case of Lesser Evil is embedded systems, thus we built a test system using popular, open-source Erlang applications to evaluate Lesser Evil. We show the test subject, present the experiments, assess the results and discuss the limitations.

5.1 Test Subject

In this section we show the representative Erlang system we built for evaluation purposes: an embedded device controller. We use the system to evaluate Lesser Evil. During the experiments we do not use representative load or scale, our goal is to push the system to its limit, because we want to confirm that Lesser Evil helps the Erlang node avoid a major outage.

The system under test (SUT) is an embedded device controller. The device controller is reachable via an HTTP API. Requests sent to the controller are being processed in memory, and confirmations are returned as response. The processing time and the memory required to process the request grow with the size of the request. Table 1 shows the processing time and the allocated memory per request size.

Table 1. Impact of an Incoming Request on the SUT

Size	Request Size	Used Memory	Processing Time
XS	1 KB	16 KB	1 ms
S	10 KB	160 KB	24 000 ms
M	100 KB	1 600 KB	48 000 ms
L	1 000 KB	16 000 KB	96 000 ms

The SUT is a one-node Erlang system built up from one application, the `http_api`, implementing the HTTP handlers. To implement the HTTP handlers we used the `cowboy` [27] and the `jsone` [30] applications as applications dependencies of the `http_api` application. The `cowboy` application depends on the `ranch` application [28] that is a socket acceptor pool for TCP protocols.

5.2 Configuration

All experiments ran on a machine that has 16 GB of memory and is equipped with a 2,6 GHz 6-Core Intel Core i7 processor. The SUT was running in a Docker container. To measure the memory consumption of the SUT we periodically queried the memory allocations of the Erlang VM using the erlang:memory/1 function provided by the Erlang VM. To generate load for the SUT, we used Basho Bench [2].

The `lesser_evil` application was deployed as a standalone Erlang node and configured to allow 90 MB memory for the SUT. The SUT packaged together with `lesser_evil_agent` was deployed as another Erlang node inside of a Docker container. The `lesser_evil_agent` application was configured to monitor and report metrics about the `ranch` application in every second. Note that the `ranch` application is not written by the authors. The Docker container running the SUT was configured to limit both real and virtual memory to 120 MB.

To generate load for the SUT, we used the following load configurations. All tests were run for 10 minutes, maintaining 5 concurrent connections. The

generated requests were random binaries, belonging to one of the request types (see Table 1). The mix of load was 20% of XS-sized requests, 40% of S-sized requests, 20% of M-sized requests, and 20% of L-sized requests. The mix of load characterises normal operation (80% of the requests are small requests) where unusually large requests (20% of the requests) occur. The only variable part in the load generation was the number of requests per second generated by each connection, that were one of the followings: $\{5, 10\}$. Observe the followings.

- The memory usage of the SUT correlates with the number and the size of requests being processed.
- The SUT can exceed the available memory when unusually large requests (L-sized requests) arrive.
- L-sized requests are the ones mainly responsible for increased memory consumption of the SUT.

5.3 Experiments

The goal of the experiments is to test the following hypotheses.

Hypothesis #1: Lesser Evil can control the memory usage of an Erlang node, therefore, it helps an Erlang node survive low memory conditions. For this purpose, we first establish a baseline: we start a test run without Lesser Evil and record the memory consumption of the SUT. Then, we start another test run with Lesser Evil and record the memory consumption. We expect to see that with Lesser Evil the memory consumption of the SUT is constrained by the given memory limit.

Hypothesis #2: Lesser Evil can prevent major outages. For this purpose, the test runs need to employ a load configuration that stresses the SUT enough to require more memory than what is available in the system. Without Lesser Evil we expect to see that the SUT will be terminated by the Linux OOM manager causing a major outage, while with Lesser Evil we expect to see that the SUT keeps operating. We accept temporary, partial system degradations.

Hypothesis #3: Lesser Evil selects and executes actions on those processes that are mainly responsible for the memory usage and does not interfere with the rest of the processes. Considering the embedded device controller, we know that the processes handling L-sized requests are the ones mainly responsible for holding memory. We expect to see that if the SUT is about to exceed the memory limit only these HTTP requests will fail. Moreover, we also expect to see that the system will continue to operate and processes executing the inexpensive HTTP requests (XS- and S-sized requests) will continue to succeed.

Hypothesis #4: Lesser Evil's agent is non-intrusive to the Erlang node, its memory usage is low. For this purpose, we first establish a baseline: we start the SUT without Lesser Evil and record its memory consumption. Then, we start the SUT packaged together with Lesser Evil and record the memory consumption. No load is generated in both cases. We expect to see that with Lesser Evil the increase in memory consumption is low.

Table 2. Experiments Conducted on the Embedded Device Controller. Note that the number of requests (# Req.s) is only shown if the experiment was successfully completed.

L.E active	Req/sec per Con.	Got OOM-ed	Max Mem.	# Req.s	# Error	# GC	# Kill
No	5	After 13 sec.	67MB	n/a	n/a	n/a	n/a
Yes	5	No	161MB	41	9	54	22
No	10	After 9 sec.	126MB	n/a	n/a	n/a	n/a
Yes	10	No	117MB	54	8	34	22

5.4 Results

The experiments were run multiple times, the test results were consistent. Table 2 summarises the experiments conducted on the test subject. It shows whether Lesser Evil was used, the maximum number of requests each connection sent per second, whether the SUT got killed by the Linux OOM manager, the maximum amount of used memory, the total number of requests the SUT received, the number of requests the SUT did not serve because of an error, the number of times Lesser Evil initiated garbage collection on a process and the number of times Lesser Evil terminated a process.

We can observe that the SUT without Lesser Evil never managed to complete any experiments; all experiments ended prematurely as the SUT ran out of memory and the Linux's OOM manager terminated the whole Erlang node. As the SUT never got OOM-ed while Lesser Evil was active and the memory usage of the SUT was 93% of the time below the configured maximum, the experiments confirm Hypothesis #1 and partially Hypothesis #2. The results suggest that Lesser Evil can control the memory usage of an Erlang node.

Figure 1 shows the memory usage of the SUT recorded during the experiment where Lesser Evil was active and each connection was configured to send 10 requests per second. Observe how effectively Lesser Evil reacts: once the memory limit (90 MB) is exceeded, Lesser Evil executes a compensating action that fixes the problem.

Based on the logs written by Lesser Evil, there were several actions executed. Both the *trigger_gc* and the *terminate_proc* actions were invoked by Lesser Evil. Actions were triggered frequently but the cool down period was always respected. As there were processes terminated by Lesser Evil, we need to confirm that the right processes were selected for termination. As only L-sized requests failed in both experiments (9 and 8 errors), we conclude that Lesser Evil chose the right processes to terminate, did not interfere with the rest of the processes, and effectively controlled the memory usage of the SUT, confirming Hypothesis #3. Due to the failed requests there was a temporary, partial system degradation that effected only the requesters of L-sized requests. However, a temporary, partial system degradation is still a better choice compared to the test runs where the SUT got terminated by the Linux OOM manager and, therefore, a major outage occurred. The experiments confirm Hypothesis #2.

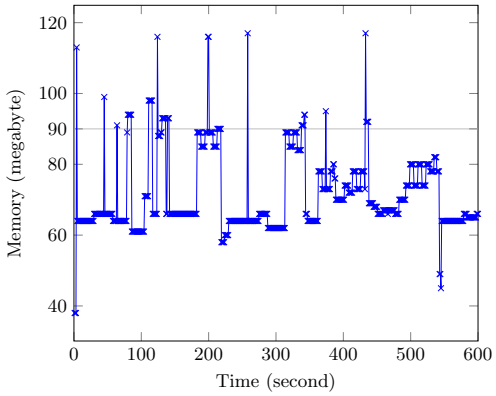

Fig. 1. Memory usage over time of the embedded device controller with Lesser Evil.

Overhead We started the SUT with and without Lesser Evil and recorded the rss size [29] for a minute. Then, we calculated the difference point by point of the recorded values; the difference we got is the memory used by the Lesser Evil agent code. The minimum, median and maximum values of memory used by the agent code are: 848 KB, 1 000 KB, 1 148 KB, which we consider low enough. The experiment confirms Hypothesis #4.

5.5 Conclusion & Limitations

The experiments confirm all 4 hypotheses. Lesser Evil can control the memory usage of an Erlang node and avoid major outage. Its negative impact on the overall system is minimal. Considering the embedded device controller use case, which is the primary use case of Lesser Evil, that includes an Erlang system with high availability but with very limited memory and slow processing capabilities, Lesser Evil can help in scenarios where a few unusually large requests arrive that take long to process and fill up the available memory.

We have experimented with other use cases. We have found that if the memory pressure is due to a continuously arriving, vast amount of short lived (ca 2 milliseconds) processes with high memory consumption, Lesser Evil cannot effectively control the memory usage of the Erlang node. To fix memory pressure, Lesser Evil would need to execute the *terminate_proc* action without respecting the cool down internal: risking to cause the termination of the Erlang node.

5.6 Threats to Validity

Our evaluation is subject to threats to validity. The results were obtained from an Erlang system, and hence, they cannot necessarily be generalized to all Erlang systems. We minimised this threat in three ways. First, by reviewing the most widespread use cases of Erlang to build a representative test system that characterises well the given use case of Erlang. Second, by building the test subject

using Erlang applications (cowboy, ranch, etc.) that are widespread and heavily used in real-life Erlang applications, and are not built by the authors. Third, by ensuring that the code written by the authors is very small (202 lines out of 267K lines) considering the size of the project, and Lesser Evil monitors an application (`ranch` [28]) that is independent from the authors.

6 Related Work

Memory pressure has always been an important challenge of the research and practitioner community. The topic has been discussed in different contexts ranging from operating systems to software libraries targeting specific software applications.

Considering operating system level, there is a generic solution for Linux systems, called the OOM manager [23]. The OOM manager is tasked to monitor the host and, under low memory conditions, select and terminate an OS process to treat memory pressure. Its goal is to protect the availability of the overall host. It employs a strategy to select new, non-user preferred processes that have high memory usage. Considering Erlang systems, our experience and the evaluation presented in this paper show that OOM is not a help, we have never seen a case when killing the Erlang node solved the underlying problem. Instead, it damaged the Erlang systems. Lesser Evil and the OOM Manager work on different abstraction level, however, both protect the overall system availability, employ a strategy and treat memory pressure by killing processes.

Virtualisation techniques have become significant to better utilise available resources and to isolate applications. An example is Apache Mesos [11, 24] that sits between the application layer and the operating system and makes it easier to deploy and manage applications in large-scale clustered environments more efficiently. Apache Mesos supports oversubscription [31] for better resource utilisation with the promise of keeping QoS metrics. Their approach is to monitor the entire host and if CPU pressure occurs the QoS controller kills revocable tasks. To further improve killing tasks when oversubscription occurs, the authors of [10] propose a user-assisted OOM killer in kernel space for agile task killing. Lesser Evil and the Mesos's QoS controller have different focus (memory versus CPU), however, both select and terminate entities that are non-critical to the system.

As there is no general solution for programming language and runtime level, researchers have proposed various solutions to better manage specific use cases [5, 7, 8, 21]. Browsing through the articles, one can notice that there has been a targeted interest in Java programs. ITask proposed in [21] is a promising choice for Java workflow applications that requires code modification, however, promises protection of tasks and performance improvements in high load scenarios. This is a considerable benefit compared to Lesser Evil, however, on the other hand rewriting applications is not required by Lesser Evil. Besides, as Erlang applications often employs restart strategies for important worker processes offering the restoration of interrupted tasks, it is not a necessity. The authors of [6] proposed

a more general solution for handling out of memory errors in Java applications. The idea is to overallocate memory that is not used and release the non-used memory when out of memory errors occur. The authors of [35] looks at the problem from another perspective, taming the garbage collection, and proposes a system that enables garbage-collected applications to predict an appropriate heap size, allowing the system to maintain high performance while adjusting dynamically to changing memory pressure.

When embedded systems are targeted, memory management becomes more critical as memory is more limited and virtual memory is often disabled. Researchers have proposed several low level memory management techniques that focus on memory allocation strategies and minimises memory fragmentation [26, 34]. The authors of [3] observed that application performance is impacted by the employed memory allocation strategy in embedded systems. They propose to manage memory per task and introduce a runtime scheduler for memory management policy switching and kernel overlapping. Their evaluation suggests that their approach can treat memory pressure and improve response time. Virtualisation ensures isolation of applications that is necessary in smart consumer electronics. However, with a virtualised embedded device, flexible memory management is required to run multiple VMs efficiently on resource-constrained hardware. The authors of [25] tackle the challenge by introducing an in-memory compressed swap device (CSW) that prioritises the memory reservations of the critical applications running on the embedded device by swapping out only the memory of third-party applications in response to memory pressure. Lesser Evil and CSW employ different compensating actions but both activate when low memory conditions occur and distinguish between critical and non-critical entities.

7 Conclusion

In this paper we have proposed and implemented an approach, called Lesser Evil, that can treat low memory pressure in Erlang systems without the need of any code modification. Lesser Evil embraces failures to protect overall system availability. It monitors the running program and upon low memory conditions selects some non-critical entities based on the badness metric and execute compensating actions on the selected entities to help the system avoid a major outage. Lesser Evil is ready to use and is applicable to any Erlang systems. The prototype implementation is available on GitHub [22].

The evaluation shows that Lesser Evil can control the memory usage of an Erlang node and an embedded Erlang system can avoid a major outage and keep functioning under low memory conditions with the help of our approach. We have confirmed that Lesser Evil's negative impact on the overall system is minimal, and Lesser Evil's strategy is correct: it is able to identify and execute compensating actions on the components most responsible for the situation without damaging the other components or the overall system.

References

1. 2021 Statistics and Data: The Most Popular Programming Languages – 1965/2021 – New Update, https://statisticsanddata.org/data/the-most-popular-programming-languages-1965-2021/
2. Basho: Basho Bench, https://github.com/alexandrejbr/basho_bench
3. Bateni, S., Wang, Z., Zhu, Y., Hu, Y., Liu, C.: Co-Optimizing Performance and Memory Footprint Via Integrated CPU/GPU Memory Management, an Implementation on Autonomous Driving Platform. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 310–323 (2020). https://doi.org/10.1109/RTAS48715.2020.00007
4. Bevemyr, J.: How Cisco is using Erlang for intent-based networking. Talk at Code BEAM STO 2018, Stockholm, Sweden, 31 May, 2018, https://youtu.be/077-XJv6PLQ?t=109
5. Borkar, V., Carey, M., Grover, R., Onose, N., Vernica, R.: Hyracks: A flexible and extensible foundation for data-intensive computing. In: 2011 IEEE 27th International Conference on Data Engineering. pp. 1151–1162 (2011). https://doi.org/10.1109/ICDE.2011.5767921
6. Boyland, J.T.: Handling out of memory errors. In: ECOOP 2005 Workshop on Exception Handling in Object-Oriented Systems. vol. 2005 (2005)
7. Bu, Y., Borkar, V., Jia, J., Carey, M.J., Condie, T.: Pregelix: Big(ger) graph analytics on a dataflow engine. Proc. VLDB Endow. **8**(2), 161–172 (Oct 2014). https://doi.org/10.14778/2735471.2735477
8. Bu, Y., Borkar, V., Xu, G., Carey, M.J.: A Bloat-Aware Design for Big Data Applications. In: Proceedings of the 2013 International Symposium on Memory Management. p. 119–130. ISMM '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2464157.2466485, https://doi.org/10.1145/2464157.2466485
9. Cesarini, F., Thompson, S.: ERLANG Programming. O'Reilly Media, Inc., 1st edn. (2009)
10. Chen, W., Pi, A., Wang, S., Zhou, X.: OS-Augmented Oversubscription of Opportunistic Memory with a User-Assisted OOM Killer. In: Proceedings of the 20th International Middleware Conference. p. 28–40. Middleware '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3361525.3361534
11. Choudhury, D.G., Perrett, T.: Designing cluster schedulers for internet-scale services: Embracing failures for improving availability. Queue **16**(1), 98–119 (Feb 2018). https://doi.org/10.1145/3194653.3199609
12. Cronqvist, M.: Troubleshooting a Large Erlang System. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang. p. 11–15. ERLANG '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1022471.1022474, https://doi.org/10.1145/1022471.1022474
13. Ericsson AB: Erlang Reference Manual: Errors and Error Handling, https://erlang.org/doc/reference_manual/errors.html
14. Ericsson AB: Erlang Reference Manual: Processes, http://erlang.org/doc/reference_manual/processes.html
15. Ericsson AB: Erlang Run-Time System Application, Internal Documentation: Erlang Garbage Collector, https://erlang.org/doc/apps/erts/GarbageCollection.html

16. Ericsson AB: OTP Design Principles: Applications, https://erlang.org/doc/design_principles/applications.html
17. Ericsson AB: OTP Design Principles: Releases, https://erlang.org/doc/design_principles/release_structure.html
18. Ericsson AB: OTP Design Principles: Supervisor Behaviour, https://erlang.org/doc/design_principles/sup_princ.html
19. Erik Stenman: The BEAM Book: Mailboxes and Message Passing, https://blog.stenmans.org/theBeamBook/#_mailboxes_and_message_passing
20. Erik Stenman: The BEAM Book: Scheduling: Non-preemptive, Reduction counting, https://blog.stenmans.org/theBeamBook/#_scheduling_non_preemptive_reduction_counting
21. Fang, L., Nguyen, K., Xu, G., Demsky, B., Lu, S.: Interruptible Tasks: Treating Memory Pressure as Interrupts for Highly Scalable Data-Parallel Programs. In: Proceedings of the 25th Symposium on Operating Systems Principles. p. 394–409. SOSP '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2815400.2815407
22. Fördős, Viktória and Barbosa Rodrigues, Alexandre Jorge: Lesser Evil, https://github.com/viktoriafordos/lesser-evil
23. Gorman, M.: Understanding the Linux Virtual Memory Manager. Prentice Hall PTR, USA (2004)
24. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I.: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. p. 295–308. NSDI'11, USENIX Association, USA (2011)
25. Hwang, J., Jeong, J., Kim, H., Choi, J., Lee, J.: Compressed memory swap for QoS of virtualized embedded systems. IEEE Transactions on Consumer Electronics **58**(3), 834–840 (2012). https://doi.org/10.1109/TCE.2012.6311325
26. Liu, D., Wang, T., Wang, Y., Qin, Z., Shao, Z.: A Block-Level Flash Memory Management Scheme for Reducing Write Activities in PCM-Based Embedded Systems. In: Proceedings of the Conference on Design, Automation and Test in Europe. p. 1447–1450. DATE '12, EDA Consortium, San Jose, CA, USA (2012)
27. Loic Hoguin: Cowboy, https://github.com/ninenines/cowboy
28. Loic Hoguin: Ranch, https://github.com/ninenines/ranch
29. Michael Kerrisk: Linux manual page: ps(1), https://man7.org/linux/man-pages/man1/ps.1.html
30. Takeru Ohta: Jsone, https://github.com/sile/jsone/
31. The Apache Software Foundation: Apache Mesos Documentation: Oversubscription, http://mesos.apache.org/documentation/latest/oversubscription/
32. Thomas, D.: Programming Elixir: Functional — Concurrent — Pragmatic — Fun. Pragmatic Bookshelf, 1st edn. (2014)
33. TIOBE Software BV: TIOBE Index, https://www.tiobe.com/tiobe-index/
34. Venkataramani, V., Chan, M.C., Mitra, T.: Scratchpad-Memory Management for Multi-Threaded Applications on Many-Core Architectures. ACM Trans. Embed. Comput. Syst. **18**(1) (Feb 2019). https://doi.org/10.1145/3301308, https://doi.org/10.1145/3301308
35. Yang, T., Berger, E.D., Kaplan, S.F., Moss, J.E.B.: CRAMM: Virtual memory support for garbage-collected applications. In: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 103–116 (2006)