Abstract—The actor model is present in several mission-critical systems, such as those supporting WhatsApp and Twitter. These systems serve thousands of clients simultaneously, therefore demanding substantial computing resources usually provided by multiprocessor and multicore platforms. Non-Uniform Memory Access (NUMA) architectures account for an important share of these platforms. Yet, little or no research has been done on the suitability of the current actor runtime environments for these machines. Current runtime environments assume a flat memory space, thus not performing as well as they could. The NUMA environment presents challenges to the actor model runtime environment in fields varying from memory management to scheduling and load-balancing. In this document we analyze and characterize application knowledge to take better memory management, virtual machine uses the NUMA characteristics and the environment, the Erlang virtual machine. This modified concept, we have applied our ideas in a real actor runtime environments for these machines. Current runtime research has been done on the suitability of the current actor model; NUMA; physical topology; Erlang;

I. INTRODUCTION

The actor model was originally proposed in the context of artificial intelligence [1] and only a few years later it began to be regarded as a possible model for concurrency [2]. In this model, every distinct execution flow is considered an actor. An actor can spawn other actors and there is no shared data between them. The only way to observe or alter the state of an actor is to send or receive a message to and from it. To communicate, each actor has a private mailbox. The messages can be sent to any other actor that, in turn, processes them asynchronously, one message at a time, at its convenience and not necessarily in the order of reception. If an actor runs out of messages to process, it may suspend its execution and keep this status until a timeout is reached or a new message is delivered to its mailbox. The delivery of messages to the actor’s mailbox is independent of its state, i.e., even if the actor is busy the delivery of new messages is not blocked.

The memory isolation, the exchange of messages and the serial processing of the messages by each actor allows for the nonexistence of locks, semaphores or any other synchronization specific tool. Actual synchronization between actors is achieved through the exchange of messages. Although powerful, this abstraction – willingly – takes the application developer away from the architectural idiosyncrasies of the machine. Thus, the actor runtime environment (RE) is responsible for the efficient use of the underlying architecture.

The actor model has some characteristics that differentiate it from other programming models. For instance, with some few exceptions, an actor lifespan is usually very short. Actors are created to perform very specific tasks and then they are discarded. Moreover, actors are frequently created in far greater quantities than the number of available processing units (PU). The reason behind it is two-fold. First, actors are, in general, mostly inactive since for most of their lives they are just waiting for messages. Second, actors keep the state of a system; there is no shared memory, so in order to access data that must be shared among many actors, for example, one needs to create an actor to hold that value. In this sense, an actor can be quite similar to an object in a object oriented programming language where an object encapsulates its state. The vast number of actors and their independence makes the actor model a good choice to take advantage of the new multi-core machines.

In 2004, as the first multi-core processor for the mainstream market was unveiled, one could already realize the new trend in processor development strategies for the following years [3]. Consequently, performance improvement has become as much of a software problem as it was, until then, an exclusive hardware problem. Nowadays multi-core and many-core processors already are the norm both on professional and personal environments. Some processors such as those from the Tilera Tile-Gx family have 72 cores each [4]. Even mobile low power consumption processors, such as Samsung’s Exynos 5 Octa [5] and NVIDIA’s Tegra 4 [6], already have more than four cores. Multi-core processors marked the general adoption of shared cache levels. While larger shared cache levels simplify the internal workings of these processors, they may also cause increased cache contention, and unpredictable variations in execution time. Furthermore, some architectures have an asymmetrical memory hierarchy,
Thus communication costs between PUs, even those in the same processor, are not constant. NUMA architecture further increases the memory hierarchy asymmetry by adding local memory to each node.

NUMA architectures have been the preferred choice of hardware developers for machines with a large number of cores. This has been motivated, in part, by the fact that these architectures are able to run regular applications developed for flat memory space architectures with no modifications. If their specific hardware characteristics are not taken into consideration, concurrency for the shared resources might cause important performance degradation [7], [8]. Efficient utilization of these machines has been a very active field of research [9], [10], [11], [12]. However, to the best of our knowledge, there has been no research on the adaptation of actor model REs for these platforms.

The actor model RE, as an additional layer over the operating system, has supplementary information about the application behavior. This information can be used by the RE to make better scheduling and load-balancing decisions. Possible strategies include: analysis of the actors communication graph to better place actors during runtime, hierarchical work-stealing and load balancing, actor pinning, and hub actors identification and placement.

In this paper, we analyze and describe actor based applications and REs. To illustrate our analysis, we present a real actor based application and RE. Our analysis covers the suitability of the current REs to the NUMA platform in order to propose a set of simple, yet quite efficient, improvements to these REs. These improvements, to the best of our knowledge, are original and therefore not being used by any actor RE (language and library based). To evaluate the relevance of our proposed modifications, we have modified the Erlang’s virtual machine (VM) [13] and assessed its performance with a set of standard benchmarks. Our tests show that, compared to the vanilla VM, our modified version achieved, in the best case, a performance improvement factor of 2.50 while limiting the slowdown on the worst case by a factor of 1.09.

The remainder of this paper is organized as follows. Section II briefly describes the actor model application characteristics and how the Erlang VM employs it. Then Section III details our proposal for the improvement of the RE followed by Section IV where we explain our evaluation methodology. Next, we show our experimental results in Section V and compare our approach to related work in Section VI. Finally, we conclude in Section VII.

II. BACKGROUND AND MOTIVATION

Actor model-based software is more common than one might think. Actor based systems are used in a wide range of applications varying from embedded systems to social networking sites. Some of the most notorious cases include Ericsson’s high-performance ATM routers [14], Facebook chat servers [15], Twitter’s messaging queues [16] and NoSQL databases [17]. In this section we analyze the general characteristics and behavior of actor based applications. To illustrate some of these aspects, we use concrete examples. Then, we discuss how these applications are actually executed by the RE.

A. The Actor Model

Some aspects of the actor model can be directly used to improve the performance of the available REs. The knowledge of the actors behavior, their communication graph, the communication costs of the target machine and the relationship between the actors are all examples of knowledge easily extractable from the RE that can be used to improve its own scheduler and load balancer decisions.

To better illustrate some of these aspects we use a real application as an example. This application, CouchDB [18], is a NoSQL database written in Erlang. We believe it is a good example to illustrate the application aspects we are interested in. The results in this section were obtained using CouchDB version 1.3.0 on the machine described in Section IV-A.

1) Actor lifespan: Not every actor is created equal. Every application has specialized actors to perform distinct kinds of work. It would be a futile exercise to try to list every possible type of actor. We can, however, define two major categories of interest and analyze their general properties. In this context we are interested in the short and long-lived actors.

Our illustrative application, CouchDB, creates many short-lived actors. Actually 99.5% of the actors live less than 1.5s and 88.9% less than 0.1s. The time needed to create a (minimal) actor is, on average, 1.5μs.

The real proportion between short and long-lived actors is application specific. We can, however, draw some conclusions about the RE from this simple example. First, the actor RE must be very efficient for the execution of short-lived actors, otherwise applications such as CouchDB would not perform well. For short-lived actors, the decision of the initial actor placement, i.e., the choice of the core in which the actor is going to be executed, must be fast. Second, the actor scheduler must be able to deal with copious amounts of actors and with their creation in bursts. The MapReduce [19] model, used by CouchDB, Riak [17] and many other actor based applications, does exactly that: it creates many short-lived actors in a short period of time.

A typical short-lived actor is (or will try to be) active for most of its life. Nonetheless, the number of alive actors in the application typically exceeds the number of available PUs making a time sharing solution a necessary part of the RE.

2) Actor Communication Costs: In principle, every communication on the actor model is based on message passing. How the actual RE does it is implementation dependent. However, it is reasonable to assume that an efficient implementation on an SMP platform will be using shared memory to provide this service. Shared memory communication costs on NUMA machines are defined not only by the size of the message but also by the location of the sender and receiver. In NUMA platforms, communication costs can easily become one of the determining factors of the application performance.
[10]. Figure 1 shows the performance penalty incurred to send messages of different sizes considering the cost between actors on the same PU, i.e., sharing the level one cache, as the baseline. For smaller messages, the inter-node performance can be more than seven times slower while for bigger messages the performance is about half that of the baseline. For the intra-node case, the performance is about three and two times worse for small and big messages respectively.

![Figure 1. Intra and inter-node performance penalty ratio associated to messages exchange of different sizes. The baseline considers the time needed to send messages between actors that lie on the same scheduler. Tests were performed on the machine described in Section IV-A. Confidence intervals (95%) are too small to be visible in this figure and are therefore not plotted.](image)

Actors that have an intense flow of communication between them and are not optimally placed may cause some undesirable effects. Beyond the longer time of communication these effects might include, for example, contention on the hardware interconnections such as the NUMA links and increased number of cache misses.

In order to show one of these effects, we created a simple artificial application (depicted in Figure 2). This application is intended to demonstrate the impact of a bad placement on the cache. During the execution, we handpicked communicating actors and deliberately placed them as close as possible (on the same core) and compared it with the second best placement (distinct core, same socket). This minimal migration caused approximately a thousand times more cache misses than the optimal placement.

3) Hub Actors: Actors usually have a well and pre-defined function when they are created. Some of these functions are naturally more requested than others, making the communication and load pattern not uniform. We call hubs those actors that exchange significantly more messages than the average actor and communicate with a wide variety of distinct actors. We define the set of the actors that exchange messages with a hub actor as the hub affinity group.

To illustrate these definitions, we have analyzed the communications graph of CouchDB. The graph depicted in Figure 3 is a representation of the communications that happened during an actual execution.

The information about the hubs and their affinity group is available to the actor RE during the execution of the application. Thus, during the definition of the migrations, the RE load balancer could take the affinity group of a hub into consideration. The intent would be to minimize the communication costs between the hub and its affinity group therefore improving the overall system performance.

B. The Erlang Virtual Machine

The development of the Erlang language and VM took into consideration several aspects of the actor model such as asynchronous message passing, private mailboxes, dynamic actor creation, and no shared data. In this section we examine some of the strategies and policies of the Erlang VM.

1) Actor Scheduler: Actor model REs are usually built to use one of two approaches: the thread-based and the event-based approaches. The main difference between the two is that in the former each actor is represented by an operating system thread or process, while in the latter each actor is represented by an internal RE data structure. In this case, the RE is responsible for the scheduling of actors and the overall system load-balancing instead of the OS. While this makes the RE more complex, it also makes it

In Erlang parlance actors are called processes, however, in order to avoid confusion, from now on we will use the term actor to refer to each Erlang VM internal process, and the term process to refer to each operating system process.
more powerful since the RE has the opportunity to perform runtime optimizations that would not have been possible otherwise.

The event-based approach is the choice made by the Erlang VM while, for example, Scala [20] gives the developer the option to choose between them. In the Erlang VM actors are represented as a simple data structure. This explains how a typical Erlang application, that has dozens (sometimes thousands) of actors, can be run efficiently on machines with just a few PUs.

The Erlang actor scheduler works by creating one OS thread for each available PU (Figure 4). These threads are called schedulers. Even though binding schedulers to the available PUs could improve performance by providing a better use of the processor caches, they are not bound by default.

Each scheduler has a run queue that keeps the runnable actors assigned to it. An actor is said to be runnable when it is not waiting for messages or any other blocking operation. When scheduled for execution, it will run until its pre-determined share of the processor runs out or it is blocked by some I/O operation. At that point, the actor will be preempted by the scheduler and put back on the run queue. The scheduler will then schedule the next actor on the run queue for execution.

During the application execution, the sizes of the queues of each scheduler might become very different from each other. Even if the queues were balanced in the beginning of the execution, each actor has distinct lifespans. Moreover, when a new actor is spawned it is assigned to the same run-queue of his parent and not all the actors have the same behavior when it comes to actor spawning.

To control this imbalance, the Erlang VM employs two strategies: work-stealing and periodical load-balancing. If a scheduler runs out of actors it will start a migration logic to steal work from the other schedulers. This would be enough to keep all the schedulers busy if there is enough work for all of them. However work-stealing by itself is not enough to ensure that each actor receives a fair share of the available PUs. Hence the VM periodically runs a load-balancing routine between the run-queues. The criteria used to determine which and how many actors will be migrated from which and to which run-queues do not include the actors affinity group or if it is a hub. Only the size of the queues and the position of the actor in that queue is taken into consideration.

There is yet another kind of imbalance which is typical of underloaded systems. In this case a strategy called compaction of load is employed. This strategy (enabled by default) will try to use as few schedulers as possible and try to minimize how often schedulers run out of work. The rationale behind it is that a small number of actors in the system makes them more susceptible to bounce between schedulers since any small variation in the number of actors might prompt a migration, for example, by the load balancer. The compaction of load works by detecting how often schedulers run out of work and, if this frequency is higher than a pre-defined threshold, the RE migrates the runnable actors to a smaller set of active schedulers. The remaining schedulers are suspended. If at a some point the active schedulers are not able to keep up with the work, some of the suspended schedulers are waken up and the load is rebalanced. The reason the RE needs this kind of behavior is related to how actors are treated. Actors are supposed to be platform agnostic, and therefore it makes no sense to bind the execution of an actor (or a group of actors) to a scheduler. On the other hand, the RE has information that can be used avoid this kind of errant behavior by, for example, trying to keep hubs affinity groups close therefore avoiding bouncing.

2) Actor Memory Management: Actors share no data, and that is exactly how the Erlang VM deals with it. Each Erlang actor has its own heap. This makes it easier to implement an efficient garbage collector since such a collector does not need to “stop the world”, inasmuch as it only needs to stop the actor on which it is working. In fact, many short-lived actors never experience a garbage collection during their lifetimes being completely discarded once they are done.

Since the heap of each actor is independent, the exchange of messages is done by copying. That is, every message gets copied from the heap of the sender actor to the heap of the receiving actor. There is, however, one exception in the Erlang VM, the binary type. This data type is used to hold binary streams. Binaries bigger than 64 bytes are allocated in a shared binary heap and they are sent in messages by reference rather than by value.

Heap allocation is an intrinsic task related to the spawning of a new actor. The heap of an actor is allocated by the scheduler of the parent actor, meaning, that the scheduler responsible for the execution of the parent is also responsible for the allocation and copying of the spawned actor’s parameters. In flat-memory space machines, the location of the allocated memory does not vary, it is always local. On the other hand, on NUMA machines, the operating system can employ several different policies to memory placement. Linux, for example, uses by default a first-touch policy. For the Erlang RE this means the
spawned actor heap location will be the node where the scheduler that created it was running. We will call this location the actor’s home node.

It is important to note that home nodes are not definitive. Take for example an actor that, for whatever reason, was migrated. During its execution it might need to grow its heap to fit new data. Often it is not possible to allocate additional memory using the same memory address and, in this case, a full heap copy to the new location must be done. If the new scheduler to which the actor was migrated is not on the same node as the actor’s home node, its home node will be changed and any RE functionality that depends on this information will need to be updated. Moreover, the cost of a simple heap growth operation that would have been proportional to the size of the heap on a flat memory space machine now depends on the current actor’s location and home node.

Some of the choices taken by the Erlang VM suggest that it was not written with the NUMA architecture in mind. This is not, however, specific to the Erlang VM. To the best of our knowledge, no other actor model (language based or library based) RE takes into consideration the NUMA aspects of the machine. In the following section, we discuss some considerations the RE might employ to become better suited to NUMA platforms.

III. A NUMA-AWARE APPROACH

NUMA platforms present challenges not only to actor model REs but also to any concurrent application. The distinct costs to access different parts of the memory cause a considerable number of problems that, among others, involve process and memory placement, scheduling, load-balancing, and memory migration. We are interested in ways to efficiently exploit these platforms using currently available REs with few modifications. In order to do that, we analyzed the behavior of some actor based applications. More specifically, we studied their communication graph and their hubs behavior.

We were looking for common patterns in the execution of these applications and the analysis of the communication graphs yielded two main conclusions. First, hub actors usually are responsible for the creation of the majority of the actors that belong to its affinity group (this heuristic will be heavily used in our approach). Second, the communication graph and, therefore, the affinity group of actors, are extremely dynamic. Trying to maintain an on-line representation of the graph or of the affinity group could bring an important overhead to the RE. We therefore propose a simpler approach based on some hints from the application developer.

Developers often have good insights into the execution characteristics of the application. They can, therefore, hint possible hubs. Hints do not change the functional behavior of applications and the RE could, at its own discretion, completely disregard them. However, the RE can also use them to help it make better decisions. Our approach works by giving the developer tools to flag the actors he believes are hubs. It can be done during the actor spawning, meaning that the developer has, at the moment of an actor creation, some evidence that the actor will be a hub. This kind of evidence can also come up during the execution. A later decision probably means that it depends on the evaluation of data that is only available during runtime. For example, actors chosen by on-line election algorithms might become hubs during the application execution, therefore changing their profile after their creation.

Our proposal is based on two main aspects of the RE, the load-balancing policy and the actor affinity maintenance. Load-balancing aims not only at maintaining every available PU busy most of the time, but also to ensure that every actor gets a fair share of the PUs time. The actor affinity maintenance tries to keep actors, and their affinity group, close together so that communication between them is fast. Sometimes these two goals may conflict. For example, the maximum actor affinity would be to place every actor together on the same PU, however that would leave the remaining PUs idle thus minimizing the load balance. We are after good trade-offs, in terms of performance, between these two aspects of the execution.

Different actor model REs face load-balancing in very different ways. Thread-based REs usually delegate the solution to the operating system. A natural solution considering that each actor is a operating system thread. On the other hand, event-based REs solve it themselves, normally using single or multiple run queues. The single-queued version essentially works by employing a thread pool that consumes work from this queue. This version has no need for a separate load-balancing logic. On the other hand, the multiple-queued version has a separate run queue for each thread. In this case, as the behavior of each actor is different, there might be some imbalance between the queues. That is why work-stealing and load-balancing algorithms are employed.

The thread-based model imposes some limits on what one is able to observe and act upon since the load-balancing decisions are taken by the operating system. We will therefore concentrate on the event-based approach. Among the event-based approaches, the single-queued one is the simplest. It works very well in a flat memory space machine with a small number of PUs. However, as the number of execution flows grows larger, the contention to access the common queue increases, thus limiting the scalability of the system [21]. Furthermore, on a NUMA platform, threads will probably be distributed throughout the whole machine. In this scenario, the common queue will distribute the actors execution evenly across the threads. This causes actors to bounce between threads, creating a significant number of cache misses therefore increasing the traffic on the NUMA interconnection. In other words, this solution does not promote soft-affinity. On the other hand, multiple-queued approaches have one queue per thread, thus, as long as the threads are bound to the PUs, soft-affinity is an intrinsic property. However, this approach has to take into consideration the eventual imbalance between the queues. It is at this point that the
work-stealing and load-balancing algorithms are put into place. A loaded system tends to have a small number of migrations therefore preserving soft-affinity. When this is not the case, compaction of load algorithms try to avoid migrations by decreasing the number of active schedulers to a minimum. These reasons compel us to believe the multiple-queued event-based is the most appropriate solution for the actor model RE on a NUMA platform.

Our proposal to keep both the load balanced and the actors affinity is centered around the RE load-balancing mechanisms. By applying the heuristic described in the beginning of this section, we can modify the actor placement and migration algorithms to do both things at once. The approach is divided in the following complementing categories.

**Initial Actor Placement** There are several possible policies to place a newly spawned actor depending on its expected behavior. Proportionally, hubs demand a lot more from the RE than their regular counterparts. They usually are among the biggest spawners in an application. Thus, it makes sense to try to spread the hubs in a way they do not need to compete for resources. On the other hand, regular actors are likely to communicate within their affinity set, so it makes sense to place them close to their hubs. We propose the use of two different initial placement policies, one for hubs and other for regular actors. Hubs should be spread throughout the available PUs, while regular actors should be placed near their hub/affinity group, on the same NUMA node. The best way to spread hubs will depend on the application behavior. For example, we could privilege communication by placing hubs close but not on the same PU (compact), or privilege resource independence by placing actors as far as possible (scatter). Both these strategies promote a good initial distribution of hubs among the available cores.

**Hierarchical Load-balancing and Work-Stealing** During the application execution, imbalances are bound to happen even with a good initial placement policy. That is why the RE needs a periodic load-balancer. Moreover, if a run queue becomes empty between the load-balancing rounds, a work-stealing solution might be employed as a temporary lightweight solution to keep the PUs busy. Both algorithms will migrate actors between run queues, however, to improve the overall performance of the system on a NUMA platform, the way a candidate is chosen for migration matters. The steps we propose to choose a migration candidate are, first, migrate actors back to their home node and, if that is not enough, migrate actors inside the same node. Only if these steps are not enough, consider the remaining actors for inter-node migration. These steps aim at keeping and restoring the proximity between actors in the same affinity group while maintaining them close to their home node and therefore their heap.

IV. EVALUATION METHODOLOGY

A. Machine

In order to evaluate the proposed modifications, we have used the NUMA machine depicted in Figure 5. This platform has 32 cores and 64 GiB of RAM equally divided in four NUMA nodes. Each node has one Intel Xeon Beckton X7560 eight-core processor running at 2.27 GHz. Each core has private L1 and L2 caches and all the cores share a common L3 (24MiB). The machine has a NUMA factor from 1.2 to 3.6. All the tests were run with Hyper-Threading disabled using the Linux operating system (kernel 3.5.7) and the GCC C compiler (4.7.2).

![Figure 5. Architecture of the NUMA machine used during our tests. This machine is composed of four nodes with eight cores per node. The last level cache (LLC) of each processor is shared by all its cores and the NUMA interconnection is a complete graph.](image)

B. Erlang Virtual Machine

We have used Erlang OTP R15B02 as the basis for our tests. The default VM represents our modified version of the VM running with all our modifications disabled. Throughout the text, we will refer to the unmodified VM as original or vanilla. The modified VM was compiled using the same default parameters of the vanilla VM on the machine described in Section IV-A.

The original VM was altered to make it possible for Erlang code mark actors as hubs. These hints can be given as an extra parameter during the actor creation or by setting a flag of a running actor. Depending on the chosen VM options, it might have no effect at all. On the other hand, given the right parameters, the modified VM will pin the execution of hub actors to a specific scheduler and avoid their migration due to load-balancing or work-stealing. Listing 1 shows the code to create a regular and a hub actor as well as the code to mark an existing actor as a hub.

```
Listing 1. “Erlang interfaces to create and set hub actors”

% Regular actor
Pid1 = spawn_opt(A_Module, A_Function, FunctionArgs, []).

% Hub actor
Pid2 = spawn_opt(A_Module, A_Function, FunctionArgs, [hub_process]).

% Marks currently running actor as a hub
erlang:system_flag(hub_process, true).
```

The initial actor placement policy of the Erlang VM is to place each newly spawned actor on the same scheduler of its parent, therefore, expediting its creation specially due to the memory allocation and heap initialization. We have modified the VM and added two new initial placement policies: compact and scatter. These policies may be applied only to hubs and, in this case, the default strategy is used for the regular actors.

NUMA factor is the ratio between remote latency and local latency.
Existing migration policies only take into account the number of actors on the queue of each scheduler and how frequently these schedulers run out of work. We have altered the VM to add a simple (our implementation only takes as input the list of NUMA nodes and the actors home nodes) hierarchical work-stealing algorithm. This algorithm was developed following the approach described in Section III.

C. Benchmarks

To evaluate the performance of our modified VM, we have used the BenchErl benchmark suite [22]. BenchErl suite has benchmarks to evaluate several different aspects of the Erlang VM. CPU-bound and Erlang language specific APIs benchmarks (such as those that test ETS tables and erlang:now/0) were removed since they are irrelevant to the aspects we want to test, i.e., those where the communication and the placement of the actors have an important role. Table I briefly describes the chosen benchmarks.

We have slightly modified the benchmarks code, only to add the hint needed to inform the VM about the hubs.

V. EXPERIMENTAL RESULTS

In this section we present the results we obtained using the NUMA-Aware approach presented in Section III. Section V-A begins presenting general performance results followed by an analysis of the performance impacting factors. Then, Section V-B compares the performances of the modified and vanilla VMs.

A. General Performance Analysis

The original Erlang VM has some optional parameters that set execution policies capable of improving the performance on NUMA architectures. To test the performance of our approach, we have taken as a baseline the benchmark execution performance without the use of any optional VM policies. We will refer to this configuration as original. However, for the sake of a fair comparison, we present, next to our results, the performance obtained by the best tuning of policies using only the options present on the the vanilla VM. We will refer to this configuration simply as default. Similarly, the performance results that made use of the policies we have proposed in this paper will be referred to as modified. Table II describes the policy options available on the original and modified versions of the VM.

Figure 6 depicts the general performance comparison between the default and modified configurations. Out of six benchmarks, modified was able to improve the performance in four of them: bang, ehb, orbit_int, and timer_wheel. Table III shows the full speedup list considering as two possible baselines the original and default configurations.

Let us start by analyzing the performance of the orbit_int benchmark. This benchmark creates a distributed hash table (DHT) in which each bucket is an actor. To add something to the DHT, an actor sends a message with the data to be inserted to the appropriate bucket. Upon the reception of this message, the bucket-actor might need to process it before storage. This can take some time thus it is done in parallel by the creation of multiple worker actors. When it is completed, additional data that must

![Figure 6. Normalized execution time of the benchmarks for two different data input sizes.](image)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>Intermediate</th>
<th>Short</th>
<th>Default</th>
<th>Intermediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>bang</td>
<td>1.48</td>
<td>1.32</td>
<td>1.10</td>
<td>1.13</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>0.92</td>
<td>0.93</td>
<td>0.92</td>
<td>0.93</td>
<td></td>
</tr>
<tr>
<td>ehb</td>
<td>1.27</td>
<td>1.14</td>
<td>1.27</td>
<td>1.12</td>
<td></td>
</tr>
<tr>
<td>orbit_int</td>
<td>2.90</td>
<td>1.35</td>
<td>2.08</td>
<td>1.23</td>
<td></td>
</tr>
<tr>
<td>serialmsg</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>timer_wheel</td>
<td>2.13</td>
<td>1.89</td>
<td>2.13</td>
<td>1.88</td>
<td></td>
</tr>
</tbody>
</table>

Table III

OBTAINED SPEEDUPS

## Table I

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Relevant Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>bang</td>
<td>Many-to-one message passing</td>
</tr>
<tr>
<td>big</td>
<td>Many-to-many message passing</td>
</tr>
<tr>
<td>ehb</td>
<td>Erlang version of the Hackbench [23] stress test for schedulers</td>
</tr>
<tr>
<td>orbit_int</td>
<td>Erlang implementation of a distributed hash table and master/worker architecture</td>
</tr>
<tr>
<td>serialmsg</td>
<td>Many-to-one and one-to-many message passing</td>
</tr>
<tr>
<td>timer_wheel</td>
<td>One-to-many message passing with timeouts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table II</th>
<th>TUNNING PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy</td>
<td>Availability</td>
</tr>
<tr>
<td>Scheduler</td>
<td></td>
</tr>
<tr>
<td>Binding</td>
<td>•</td>
</tr>
<tr>
<td>Compaction of Load</td>
<td>•</td>
</tr>
<tr>
<td>Initial Placement</td>
<td>•</td>
</tr>
<tr>
<td>Work-Stealing</td>
<td>•</td>
</tr>
</tbody>
</table>

![Table II](image)
be stored on the DHT might have been generated. Their generators, the worker actors, then send these data back to their master that in turn forwards them to the appropriate buckets for storage. The benchmark measures how long the RE takes to insert a specific set of data into the DHT, including the time needed to process it and insert any additionally generated data. This benchmark has clearly defined communication hubs: the buckets. These actors are involved in most communications and at the same time they perform the role of a master actor that spawns several worker actors. Although in a somewhat different way, ehb has these same characteristics that reflect the fundamental ideas behind our proposal. The significant speedup we have obtained with these applications show that when our assumptions fit the target application behavior, our approach has an important impact in performance.

The two benchmarks for which we could not improve performance, big and serialmsg, deserve more attention. These are peculiar benchmarks that specifically test the RE against extreme communication situations. The big benchmark creates a number of actors that send, all at the same time, messages to every other actor on the system. The benchmark evaluates how long the RE takes to deliver every sent message. The communication graph is a full graph. There is no communication hub that stands out and the affinity group of each actor is composed of every other actor in execution. On the other hand, serialmsg has only one communication hub. This actor acts as a message dispatcher for every other actor on the system. The benchmark works by creating two sets of actors, the senders and the receivers. The communication between these two sets of actors is done, exclusively, through this dispatcher. These two benchmarks present the RE with situations that do not fit the assumptions of our proposal. Examples of these kind of situations include those where every actor of a system is a hub, where no actor is a hub, or when there is just one hub. Our proposal assumes the application will have a few communication hubs and that we will be able to spread their affinity groups throughout the NUMA nodes. When the application has only one communication hub and its affinity group is the whole set of actors, our approach ends up introducing overheads that we are not able to compensate. In these cases, a simpler and lighter approach, such as that of the RE with situations that do not fit the assumptions of our proposal, is more suitable.

Some factors influence some benchmarks much more than others. This explains why some of them performed substantially better despite the fact that the number of migrations was kept practically constant. Some benchmarks, such as ehb are much more susceptible to alterations on the initial placement than to alterations on scheduler bindings. On the other hand, benchmarks like bang are more influenced by the scheduler bindings than by any other policy. Table IV shows the average reduction of the execution time for each of the evaluated policies using the short data set.

![Figure 7. Average number of actor migrations per execution using the intermediate data set with the VM’s default and modified configurations.](image)

Table IV

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>bang</th>
<th>big</th>
<th>ehb</th>
<th>orbit_int</th>
<th>serialmsg</th>
<th>timer_wheel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td>35.1%</td>
<td>28.7%</td>
<td>10.5%</td>
<td>53.3%</td>
<td>-13.4%</td>
<td>-81.4%</td>
</tr>
<tr>
<td>Compaction</td>
<td>-0.3%</td>
<td>0.2%</td>
<td>-0.6%</td>
<td>0.8%</td>
<td>0.1%</td>
<td>-0.6%</td>
</tr>
<tr>
<td>of Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial</td>
<td>20.5%</td>
<td>38.9%</td>
<td>20.5%</td>
<td>64.1%</td>
<td>-8.8%</td>
<td>-73.3%</td>
</tr>
<tr>
<td>Placement</td>
<td>-14%</td>
<td>21.4%</td>
<td>5.8%</td>
<td>43.3%</td>
<td>2.9%</td>
<td>-22.0%</td>
</tr>
<tr>
<td>OnlyHubs</td>
<td>16.9%</td>
<td>21.4%</td>
<td>5.8%</td>
<td>43.3%</td>
<td>2.9%</td>
<td>-22.0%</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>9.6%</td>
<td>18.1%</td>
<td>9.8%</td>
<td>43.2%</td>
<td>-5.4%</td>
<td>-37.6%</td>
</tr>
<tr>
<td>Work-Stealing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The timer_wheel benchmark is an interesting case by itself. Every policy change caused performance degradation. Actually, the performance gains we show for this benchmark were obtained with every, original and proposed, alternative policy disabled. The reported speedup comes from the fact that, even with every policy disabled, our modified VM strives not to migrate hub actors. The considerable decrease on the number of actor migrations (Figure 7) is responsible for the reduction of 53% (small) and 47% (intermediate) on the reported execution time.

The compaction of load policy almost did not change the overall execution time of the benchmarks. This was, however, to be expected. Every benchmark we executed creates many more actors than the available number of PUs. This renders the compaction of load essentially inactive for most of the time.
so that it can effectively work on large scale multi-core systems. This project has been, however, until now focused on the distributed aspects of the RE.

On the other hand, the research community has shown a strong interest for non-actor concurrent REs. Rashiti [29] et al. show how a better match between the communication and physical topologies on MPI applications can bring considerable gains in communication performance. Charm++ on NUMA machines face many of the same problems of actor-based REs and it has been shown that [30] NUMA awareness might bring improvements on the overall system performance.

Improvements on the REs are not the only possibilities currently in exploration. Kernel level solutions such as AutoNUMA and NumaSched [12] try to improve the performance of the processes transparently doing better memory allocation and scheduling distributions. In particular, NumaSched also has the notion of a process home node. A process will allocate memory preferentially from its home node unless load-balancing dictates otherwise. In this case, a migration may end up changing the home node of the process. This will be followed by a lazy memory page migration. NumaSched, on the other hand, employs a different approach. For each process the Kernel maintains the last NUMA nodes of the memory pages it has recently accessed. Similarly, for each page the kernel maintains the last NUMA node that accessed it. Based on these statistics, the kernel decides if (and where) a process or a memory page needs to be migrated. Unfortunately, this kind of approach has limited efficiency on REs (for the actor model or not) that do not have a direct link between each internal flow of execution and an operational system thread or process. Furthermore, the actor RE has additional higher level information, opaque to the Kernel, that can be used to make better scheduling and memory allocation decisions.

Our proposal involves a programming methodology (actor model) and the scheduling and load-balancing mechanisms necessary to make it more efficient on NUMA multi-core platforms. We have applied our proposal to a specific RE, however, it is not directly related to any particular programming language or RE. We strongly believe that the techniques applied to this particular environment are applicable to other actor REs. While our proposal aims at being transparent to the user, we still give him the possibility to hint hubs. These hints are not obligatory and do not alter the application functional behavior.

VI. RELATED WORK

The research involving scheduling of actors to better use multi-core machines has mostly been about the development of SMP capable REs [24], [21] and the analysis and optimization of their performances [25], [26]. These works focused on the evaluation of the available implementation choices such as single or multiple run-queues, lock-free message passing, software transactional memory, etc. There has also been some research for the development of alternative actor based concurrency frameworks [27]. However, to the best of our knowledge, there has not been any research that took into consideration the hierarchical topology of the memory (intrinsic to NUMA platforms) to implement a NUMA-Aware actor RE.

The Release Project [28] aims to create a high-level paradigm for reliable large-scale server software. Among others, one of their goals is to evolve the Erlang VM

B. Original VM Comparison

The performance results presented in Section V-A, although measured using the default behavior, were assessed using the modified VM. This was done not only due to the limitations of the original VM (for example, there is no support to trace actor migrations), but also because we are interested in comparing the impact of each distinct policy only. We did not fine tune the modified VM code. Thus, the comparison between the modified VM performance and that of the heavily optimized original VM would defeat the purpose of our experiment. We have, however, estimated the overhead our modified code imposes to the execution of the benchmarks. Our measurements show an overhead ranging from 2% to 26%. Such an overhead range allows us to, in some cases, employ the modified VM and obtain significant performance gains even without the code optimizations. Figure 8 depicts the execution of the benchmarks for two distinct workloads. We show for each benchmark the performance of the best tuning for the original and modified VMs. Execution times were normalized by the original VM execution time with no optional policy parameters.

VI. RELATED WORK

The research involving scheduling of actors to better use multi-core machines has mostly been about the development of SMP capable REs [24], [21] and the analysis and optimization of their performances [25], [26]. These works focused on the evaluation of the available implementation choices such as single or multiple run-queues, lock-free message passing, software transactional memory, etc. There has also been some research for the development of alternative actor based concurrency frameworks [27]. However, to the best of our knowledge, there has not been any research that took into consideration the hierarchical topology of the memory (intrinsic to NUMA platforms) to implement a NUMA-Aware actor RE.

The Release Project [28] aims to create a high-level paradigm for reliable large-scale server software. Among others, one of their goals is to evolve the Erlang VM
time a NUMA-Aware RE for the actor model is proposed. To assess the impact of our optimizations, we modified a real actor RE and evaluated its performance using standard benchmarks. Our experiments show that our NUMA-Aware optimization policies can bring significant performance improvements to actor model REs.

While the obtained optimizations results are relevant, some simplifying assumptions were made for the development of our prototype. The most important are:

**NUMA Interconnection** For the sake of model simplification, we have assumed that all the nodes of a NUMA platform are interconnected. Although our test machine displays this characteristic, this is not true for every NUMA machine. Platform-specific distance functions and the appropriate RE adaptations might bring better performance improvements in these cases.

**Hubs, Affinity Groups and Home Nodes** Our assumption that hub actors are, for most of the time, responsible for the creation of the majority of the actors that belong to its affinity group is very strong and may not be true for every application. Moreover, home nodes can be changed during the execution. Our approach uses the current location and the actor home node as an indication of the proximity to its hub and, therefore, affinity group. By removing the need to use such heuristic, we expect to improve the effectiveness of our approach.

### References


