Improving the Performance of Actor-Based Programs Using a New Actor to Thread Association Technique

Fahimeh Rahemi, Ehsan Khamespanah, Ramtin Khosravi

Improving the Performance of Actor-Based Programs Using a New Actor to Thread Association Technique

Fahimeh Rahemi[1], Ehsan Khamespanah[1,2], and Ramtin Khosravi

[1] School of Electrical and Computer Engineering, University of Tehran - Iran
[2] School of Computer Science, Reykjavik University - Iceland

Abstract. Finding the most efficient policy for the association of objects with threads is one of the main challenges in the deployment of concurrently executing objects, including actors. For the case of actor-based programs, libraries, frameworks, and languages provide fine tuning facilities for associating actors with threads. In practice, programmers use the default policy for the initial deployment of actors and the default policy is replaced with some other policies considering runtime behaviors of actors. Although this ad-hoc approach is widely used by programmers, it is tedious and time-consuming for large-scale applications. To reduce the time-consumption of the ad-hoc approach, a set of heuristics is proposed with the aim of balancing computations of actors across threads. This technique results in performance improvement; however, it relies on the static analysis of source codes and actors' behaviors, ends in the inappropriate configuration of systems in distributed environments. In this paper, we illustrate conditions that the proposed heuristics do not work well and propose a new approach based on the runtime profile of actors for better association of actors with threads. We also show how this approach can be extended to a fully self-adaptive approach and illustrated its applicability using a set of case studies.

Keywords: Actors, Thread Association, Self-Adaptive Algorithm, Runtime Analysis

1 Introduction

The actor model is a well-known model for the development of highly available and high-performance applications. It benefits from universal primitives of concurrent computation [1], called actors. Actors are distributed, autonomous objects that interact by asynchronous message passing. This model was originally introduced by Hewitt [2] as an agent-based language and is later developed by Agha [1] as a mathematical model of concurrent computation. Each actor provides a number of services, and other actors send messages to it to run the services. Messages are put in the mailbox of the receiver, the receiver takes a message from the mailbox and executes its corresponding service. A number of

programming languages and libraries are developed for actor-based programming, e.g. Act [3] and Roset [4] which are discontinued and Erlang [5], Salsa [6], and Akka [7] as actively supported programming languages and libraries.

In the actor programming model, a large-scale distributed applications are developed by spawning many actors which are distributed among some computation nodes and work in parallel. Using this approach, utilizing CPUs of different nodes is crucial, needs careful mapping of actors to nodes and CPUs. Some of actor-based programming languages handle scheduling of actors on different cores on runtime, using a shared pool of threads for actors which are scheduled on CPUs by round-robin approach, including Erlang [8] and Kilim [9]. However, in the majority of the JVM-based actor languages, it is the duty of programmers to associate actors with threads, including Akka and Scala [10]. This way, a programmer has to associates actors with threads using the default mapping and iteratively tune the mapping, which is a very hard job and sometimes impossible for large-scale applications.

Recently, Upadhyaya et al. in [11] proposed some heuristic for the association of actors with thread. To this end, they defined an Actor Characteristics Vector (cVector) for each actor to approximate the runtime behavior of it. The details of this approach are presented in Section 2. Using cVector, actors are associated with threads using one of the predefined policies of thread-pool, pinned, and monitor policies. The main goal of this approach is to map actors to threads in a way that balances actor computational workloads and reduces communication overheads. They implemented the technique for Panini and achieved on average 50% improvement in the running times of program over default mappings [12].

Although this approach improves CPU utilization of nodes significantly, it does not take the runtime behavior of systems into account. This limitation results in inefficiencies in the performance of actor systems, particularly in cases where actors are distributed among different nodes. In this work, we address both the number of spawned actors from a specific type and the load of systems at runtime to propose a better thread association policy. To this end, we propose a new light-weight technique for capturing the runtime behavior of actors (Section 3). We show how characteristic vectors of actors have to be modified to make them appropriate for presenting runtime behaviors of actors. Also, we show how the newly proposed characteristic vector is changed during the time and thread policies of actors have to be adapted to these changes. We develop a set of case studies to illustrate the applicability of this work in Section 4.

2 Static Association of Actors with Threads

Actors as loosely coupled parallel entities have to be associated with threads to be allowed to pick messages from their message boxes and execute them. Dedicating one thread to each actor is the simplest approach for this purpose; however, as actor-based applications usually spawn many actors, this approach does not scale. To resolve this limitation, actor libraries provide different policies for allowing programmers to associate a shared thread with multiple actors.

Using this resolution, finding the appropriate policy for the association of a thread with a group of (or one) actors is the responsibility of programmers. Generally, three different types of policies for the actor with thread association is provided to cover the requirements of applications, called thread-pool, pinned, and monitor policies. The details of these policies are presented below.

2.1 Policies for the Association of Actors with Threads

The default and widely used policy for the thread to actor association is the thread-pool policy which uses a thread-pool with a limited number of threads for a set of actors. Usually, the number of actors is more than the number of threads and actors compete for threads. This policy efficiently works for actors which are not always busy, so the less number of threads can be shared among actors. Using thread-pool policy, there is no thread preemption while an actor is busy with executing a message and actor lose its associated thread only when finishes serving a message.

As the second alternative, using the pinned policy, an OS level thread is dedicated to an actor. This policy efficiently works for busy actors, so the overhead of frequently changing the associated thread with a pinned actor is eliminated.

Finally, the monitor policy is used for actors which perform very light activities. Using the monitor policy, the associated thread with the sender of a message is reused by the receiver actor to serve the recently sent message. When serving the message is finished, the actor gives back its associated thread to the sender of the message. Note that the associated thread with the sender actor only can be reused when both of the sender and receiver are deployed on the same node.

These three policies are provided by different actor libraries with different naming. Akka provides PinnedDispatcher, BalancingDispatcher, and CallingThreadDispatcher to realize pinned, thread-pool, and monitor policies. Akka also provides a default dispatcher which is a realization of thread-pool policy configured with a set of general purpose values. In contrast, the scheduler of Erlang only provides thread-pool policy. Kilim as the provider of very light Java actors only provides the thread-pool policy which is implemented efficiently to be able to handle thousands of actors.

2.2 Using Characteristics Vector of Actors

As the only work which tries to propose appropriate policies for actors, Upadhyaya et al. in [11] proposed a heuristic-based technique for setting policies of actors (henceforth, Static-Heuristic approach). In this approach, they defined the notation of Actor Characteristics Vector (cVector) for each actor to approximate the runtime behavior of that actor. They benefit from Actor Communication Graphs (ACG) of systems to generate cVectors. The vertices of ACG are actors of a system and there is an edge between two vertices if and only if there is a possibility of sending a message from an actor which is associated with the source vertex to the actor which is associated with the destination vertex. They

also marked actors which have blocking I/O activities, actors which are computationally intensive, and actors which have many communications. As a result, cVectors of actors are created as defined below.

Definition 1 (Characteristics Vectors). *Set* $\mathbb{CV}$ *as the set of the characteristics vectors of actors is defined as* $\mathbb{CV} = \{\langle blk, state, par, comm, cpu\rangle \mid blk \in \{true, false\} \wedge state \in \{true, false\} \wedge par \in \{low, med, high\} \wedge comm \in \{low, med, high\} \wedge cpu \in \{low, high\}\}$. □

For a given characteristic vector $\langle blk, state, par, comm, cpu\rangle$ for the actor ac, the interpretation of the terms is as the following:

- the value of blk is $true$ if ac represents blocking behavior,
- the value of $state$ is $true$ if at least one of the state variables of ac is accessed by more than one of its methods,
- the value of par is low if ac sends a synchronous message and waits for the result. It is $high$ if ac sends an asynchronous message and does not require result. Otherwise it is med,
- the value of $comm$ is low if ac does not send message to other actors. It is $high$ if ac sends message to more than one actor. Otherwise it is med,
- the value of cpu is $high$ if ac represents computational workload, i.e. having recursive call, loops with unknown bounds, or making high cost library calls.

Using this interpretation, function $CV : \mathbb{AC} \rightarrow \mathbb{CV}$ maps a given actor to its corresponding cVector. Here, we assumed that $\mathbb{AC}$ is the set of actors of a system. Note that [11] does not provide a precise guideline for detecting high cost library calls and blocking behavior.

To map a cVector to a thread policy, a function is defined in Definition 2. This heuristic states that a thread has to be associated with an actor (pinned policy) that has external blocking behavior. Any other policy for these actors would lead to blocking of the executing thread and may lead to actor starvation or deadlocks. In addition, any actor that is non-blocking with high inherent parallelism, high communication, and high computation should be assigned the pinned policy. Master actors, which have the property that they delegate the work to slave actors and often wait for the result are eligible for the pinned policy.

Actors with low CPU consumption and communication do not need special attention and hence are processed by the calling actor (the actor that sends messages). Actors with other characteristic vectors can share their associated threads; so, the thread-pool policy is assigned to them.

Definition 2 (Mapping cVector to Policy). *An actor which corresponds to the cVector* $cv \in \mathbb{CV}$ *is mapped to a thread policy by function* $HF : \mathbb{AC} \times \mathbb{CV} \rightarrow \{pinned, thread\text{–}pool, monitor\}$ *where:*

- $HF(cv) = pinned$*: if and only if* cv *is in the form of* $\langle true, -, -, -, -\rangle$, $\langle false, -, high, high, high\rangle$, *or* $\langle false, -, low/med, high, low\rangle$,

- $HF(cv)$ = *monitor: if and only if* cv *is in the form of* $\langle false, -, -, low/med, low \rangle$,
- $HF(cv)$ = *thread-pool:* cv *does not fit the above cases.*

□

Note that in this mapping, being stateful/stateless does not matter.

3 Runtime Association of Actors with Threads

Although the Static-Heuristic approach for the association of actors with threads results in performance improvements, it does not consider the runtime behavior of the system. This way, both over-approximation and under-approximation of the behavior of system is inevitable and causes inefficiencies in runtime. In the following we illustrated this phenomenon and proposed a runtime approach (henceforth Adaptive-Heuristic approach) to resolve it. In addition, we showed that thread association policy is widely influenced by the deployment strategy of the application and the number of hosts of actors. So, for an efficient thread association policy, deployment strategies have to be taken into account.

3.1 Redefinition of Actors Characteristics

Performing a number of experiments, we found that two terms of cVector have to be redefined. Using the current definition, these two terms misleads heuristic in actors to threads association approaches. The first is the term that shows the level of communication among actors. As mentioned before, based on the definition of [11], the value of this term in the cVector of an actor is set to High if the actor sends more than one message to other actors. However, sending messages is a very light operation which is not affected by thread policies. Instead, level of communication has to be set to High for an actor which receives many messages. Many received messages results in needs for many future computational power, which is tightly in relation with thread policies. To make this difference clear, we use the example of hub-actor in [11]. Hub actors are represented by either $\langle false, , high, high, low \rangle$ or $\langle false, , low/med, high, high \rangle$ which show that they have high communication characteristics. It is because of the fact that the affinity actors (actors that hub actor communicates often) send message to the hub actor, which is in contrast with the proposed metric in [11], i.e. sending many messages from a hub actor to the others result in High value for the communication level.

The other case which results in having high communication is receiving messages from actors which are developed in some other nodes. As we will show later, actors with high communication are not allowed to be mapped to the monitor policy which is essential for high-performance processing of messages which are sent from actors which are hosted by the other nodes. Note than the new definition addresses the runtime behavior of systems, so it can not be used in the approach of [11].

The second term that has to be redefined is the needed computational power, addressed by CPU. The needed computation power is a runtime metric which can not be effectively estimated by static analysis. Note that this argument is valid for complex actor-based systems, since the needed computational power of simple actor models can not be estimated by having a quick look into their source codes. In the new definition, the value of CPU is related to the average consumed processor time by actor. Note that the new definition sets the needed computational power for actor types not actor instances.

In addition to modifying the definition of these two terms, we found that lifetime of actors has a significant influence in the runtime behavior of actors and has to be included in the cVectors of actors. For example, using Aggregator pattern [13], a task is split into some very simple subtasks, delegated into newly instantiated actors. The newly instantiated actors complete their associated subtasks, send the result to the owner actor and die. Regardless of the values of the others terms of the corresponding cVector, these short-lived actors are very good candidates for being associated to the monitor policy. This way, one thread is used for performing all the simple subtasks and the overhead of releasing and reclaiming thread for doing subtasks is eliminated. Note that in this case we assumed that all of actors are deployed in the same computational node. Delegating threads using monitor policy is impossible when sender and receiver actors are deployed in different computational nodes.

Based on these changes, runtime characteristics vector (rcVector) of an actor is defined as the following. We still have no observation on the effect of being stateless/stateful, so we eliminate it in runtime characteristics vectors.

Definition 3 (Runtime Characteristics Vectors). *Set* $\mathbb{RCV}$ *as the set of the runtime characteristics vectors of actors is defined as* $\mathbb{RCV} = \{\langle blk, par, comm, cpu, lt\rangle \mid blk \in \{true, false\} \wedge par \in \{low, med, high\} \wedge comm \in \{low, med, high\} \wedge cpu \in \{low, high\} \wedge lt \in \{low, high\}\}$. □

The interpretation of the terms in a given rcVector $\langle blk, par, comm, cpu, lt\rangle$ for actor ac for the terms blk and par is the same as them the original characteristics vectors and for the other three terms is as the following:

- the value of $comm$ is low if the number of received messages per a unit of time of ac is less than this value in average case of all actors. It is $high$ if that value is bigger than the average, and otherwise it is set to med,
- the value of cpu is $high$ if the value of the needed computational time per method of ac is bigger than this value for the average case, considering all of actors. In the case of receiving messages from actors, deployed on the other computation nodes of the system, the value of cpu is set to $high$ too. otherwise it is set to low,
- the value of lt is $high$ if the lifetime of ac is bigger than the average lifetime of all of the existing actors. otherwise it is set to low.

Using this interpretation, function $RCV : \mathbb{AC} \rightarrow \mathbb{RCV}$ maps a given actor to its corresponding rcVector. To map a rcVector to a thread policy, a function is defined as below.

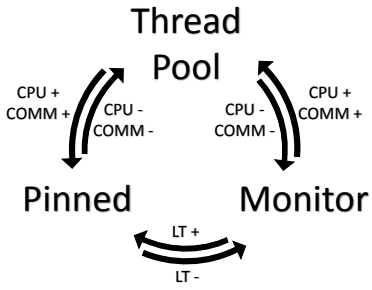


Fig. 1. An overview of the thread policy adaptation algorithm

Definition 4 (New Mapping Algorithm). *An actor which corresponds to the rcVector* $rcv \in \mathbb{RCV}$ *is mapped to a thread policy by* $RHF : \mathbb{AC} \times \mathbb{RCV} \rightarrow \{pinned, thread\text{–}pool, monitor\}$ *where:*

- $RHF(rcv) =$ *pinned: if and only if* rcv *is in the form of* $\langle true, -, -, -, - \rangle$, $\langle false, -, high, -, high \rangle$, *or* $\langle false, -, -, high, high \rangle$,
- $RHF(rcv) =$ *monitor: if and only if* rcv *is in the form of* $\langle false, -, low, low, low \rangle$,
- $RHF(rcv) =$ *thread-pool: other* $rcvs$.

□

3.2 Towards a Self-Adaptive Approach

Using runtime mapping algorithm improves performance of systems but there is an open question on how the actors must be configured at their instantiation point. It is clear that before running systems communication level, CPU consumption, and lifetime of actors are unknown, so finding the appropriate mapping is impossible for almost all of the actors (except for actors with blocking behaviors). Therefore, a default thread policy must be assumed for all of the actors and it must be changed during the execution of the system. This adaptation is crucial for making the runtime approach possible. To this aim, we propose the adaptation algorithm which is presented in Figure 1. Actors are initially use thread-pool policy and change their thread policy upon detecting any permanent changes in the values of communication level, CPU consumption, and lifetime of their rcVectors. The labels of arrows in Figure 1 shows that which changes trigger that possible adaptation. For example, "CPU +" label on arrow between thread-pool and pinned shows that for actors which thread-pool policy increasing the value of CPU results in changing the policy to pinned. Performing this adaptation, after some amount of time the system meets its high-performance steady state.

In addition to resolving the initial mapping of actors to thread policies, the adaptation policy helps in resolving inefficiencies, caused by changes in the load profile of systems (e.g. changes in the number of clients, the operational servers,

etc.). Runtime changes in the load profile of a system my change the characteristics of an actor during the time. So, some adaptation may needed after such changes to find the new high-performance steady state. The same argument is valid for actors migration, i.e. changing host nodes of actors. Based on the proposed mapping algorithm, actors migration significantly influences association of monitor policy with actors.

4 Experimental Results

To illustrate the applicability of this work we prepare some case studies and show how using the Adaptive-Heuristic approach improves the performance of systems. The presented case studies are partitioned in two parts. The first part contains a number of models which are proposed in [9]. The second part contains an example which shows runtime changes in load profile and the number of actors. We illustrate how the new approach adapts policies based on the encountered changes.

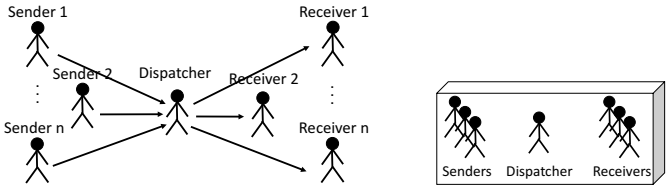
4.1 Models Without Runtime Adaptation

We use some of the models proposed in [9] and develop a simulator for pure actor programs. For the design of the simulator we consider both multi-node and multi-processor environments. This way, a number of threads are spread among nodes and each node schedules its own threads using its associated processors. Using this simulator, the models are developed without need for dealing with the complexities of the real-world Java actor programming. In addition, having simulator, we run models in different infrastructure configurations and monitor pure impact of thread association policies to the runtime execution of models.

In the following, we present an intuitive description and deployment diagram for each model. We also present a figure which compares the termination time of the model for three cases of using default thread-pool policy, the Static-Heuristic approach, and the Adaptive-Heuristic approach. The best approach has the smallest termination time, as it consumes the provided computation power better that the others.

Request Dispatcher. We develop RequestDispatcher example, i.e. message routing among a set of senders and receivers. This model contains three different actors which are Sender, Receiver, and Dispatcher. Sender actors pass messages to the Receiver actors via Dispatcher. The actor model of RequestDispatcher is shown in Figure 2.

As presented in [11], based on the characteristics vector of the actors, the Static-Heuristic approach maps Sender and Dispatcher actors to the thread-pool policy, and Receiver to the monitor policy. This mapping only works for single node deployment of actors as upon deploying Dispatcher and Receiver in different nodes, there is no way for sharing Dispatcher threads with receivers. In addition, heavy weighted receivers may block dispatchers and reduce the performance of the system. The Adaptive-Heuristic approach proposes changing the policy



(a) The actor model of RequestDispatcher example

(b) The deployment diagram of the example

Fig. 2. The RequestDispatcher example

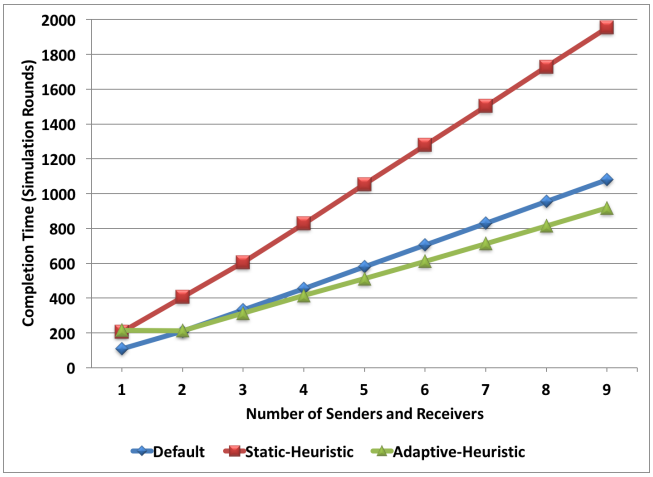
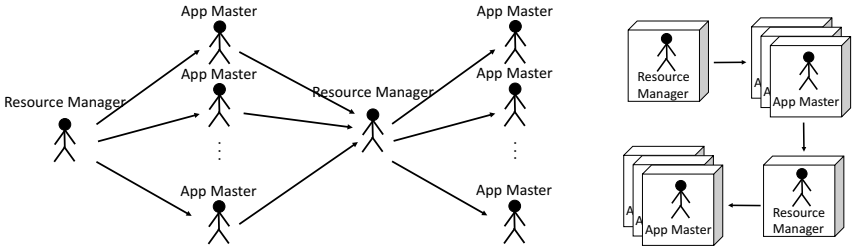


Fig. 3. The completion time of the request dispatcher model in different configurations

of Dispatcher to the pinned policy and the policy of Receiver to thread-pool. Dispatcher as the bottleneck of the model, has to be able available permanently; so, a thread has to be associated with it. Also, in the case of deploying Receivers and Dispatcher in different nodes, there is no need for changing the policy of receivers, as they do not reuse the thread which is associated with Dispatcher. Changing the number of senders and receivers resulted in the following figure for the completion time of the model.

Two Level Hadoop Yarn Scheduler. Hadoop is a framework for MapReduce, a programming model for generating and processing large data sets [14]. MapReduce has undergone a complete overhaul in its latest release, called MapReduce 2.0 (MRv2) or YARN [15]. The fundamental idea of YARN is to split up the major functionalities of the framework into two modules, a global Resource Manager and per-application Application Master. On a Hadoop cluster, there is a single resource manager and for every job there is a single application master. In this example, we modeled a pipeline of two instances of MapReduce clusters, depicted in Figure 4.



(a) The actor model of MapReduce example

(b) The deployment diagram of the example

Fig. 4. The MapReduce example

Based on the characteristics vector of the actors, the Static-Heuristic approach maps ResourceManager actors to the pinned policy, and ApplicationMaster actors are mapped to the monitor policy. However, the work load of the second ResourceManager and ApplicationMasters are shaped by the first ResourceManager and ApplicationMaster. The Adaptive-Heuristic approach proposes pinned policy at the starting point of the first ResourceManager and changes it to thread-pool in some configurations. Based on the light weight load of the second ResourceManger, the adaptive policy proposes monitor policy for this actor. Comparison among completion times of the model in different configurations is depicted in Figure 5.

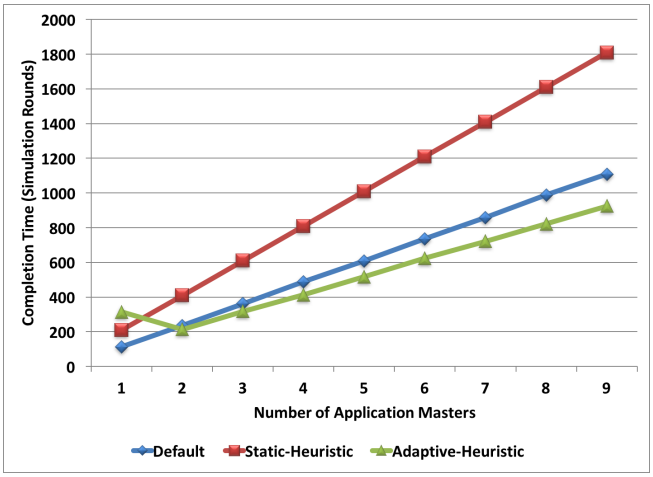
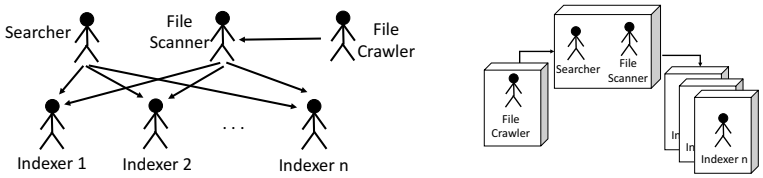


Fig. 5. The completion time of the Hadoop Yarn Scheduler model in different configurations

File Search. Document indexing and searching model [11] is the third case study that we developed. This model contains four different actors which are FileCrawler, FileScanner, Indexer, and Searcher. FileCrawler periodically visits directories which their paths are given at the start point and sends a message to FileScanner upon finding a newly modified file. To increase the verity of the number of actors in this model, we used only one crawler actor. FileScanner processes the given file and asks one of the available Indexers to index the file. Indexer performs hash-based indexing and stores the extracted information. The Searcher actor serves the search request which are sent by users. The actor model of FileSearch is shown in Figure 6(a)



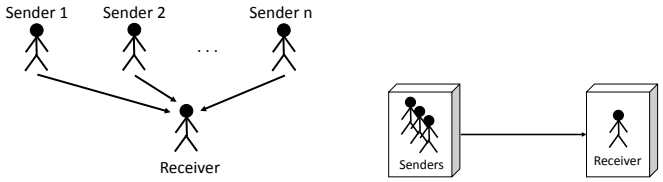
(a) The actor model of FileSearch example (b) The deployment diagram of the example

Fig. 6. The FileSearch example

As presented in [11], the Static-Heuristic approach maps FileCrawler and Searcher actor to the pinned policy, Indexer to the monitor policy, and FileScanner to the thread-pool policy, based on the characteristics vector of the actors. The same as the previous example, this mapping only works for single node deployment of actors. The Adaptive-Heuristic approach proposes changing the policy of Indexer to the pinned policy. Also, in case of deploying FileCrawler in the node which contains FileScanner, it proposes changing the policy of FileCrawler to the thread-pool policy, as there is no need for associating one thread for its periodic behavior. Experiments showed that there is a very light improvements in using the new approach.

Bang Model. The last model we developed is the Bang benchmark which simulates many-to-one message passing. As shown in Figure 7(a), in this model there is one receiver and multiple senders that flood the receiver with messages. Based on the CVector of actor, the receiver actor is mapped to the monitor policy and senders are mapped to the tread-pool policy, using the Static-Heuristic approach.

The results of [11] shows that the Static-Heuristic approach improves the performance of the system in comparison to the default policy but it does not provide the best mapping. Assume that these actors are deployed as shown in Figure 7(b). In this configuration, mapping receiver to monitor does not result in reusing the threads of senders as the actors are deployed in two different machines. In this case, the receiver actor has to be mapped to the pinned policy to



(a) The actor model of the Bang example

(b) The deployment diagram of the example

Fig. 7. The Bang example

be able to process the requests upon receiving their corresponding messages, as made by the new approach. However, experiments showed that there is no difference between using the Static-Heuristic approach and the adaptive-Heuristic one (based on the deployment of Figure 7(b)) as, there is no thread interference between the senders and the receiver.

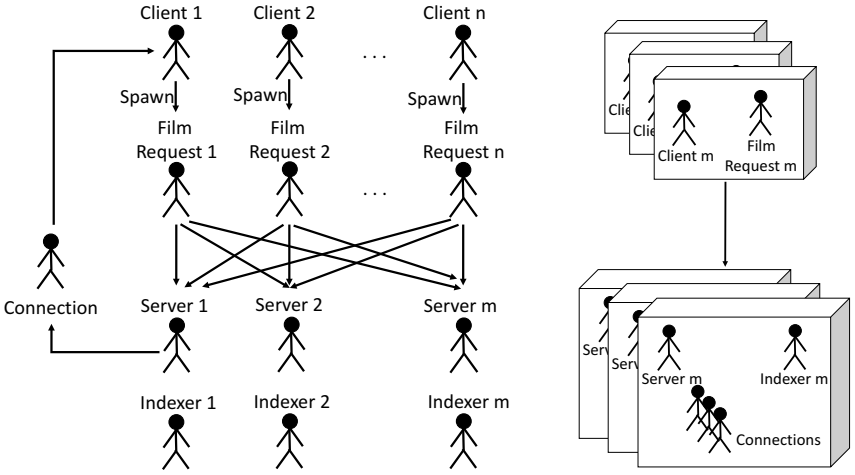
4.2 A Model With Runtime Adaptation

We presented an example in the second part of experimental result which is the model of a FilmService system, shown in Figure 8(a). In this example, clients want to stream a movie from film servers. A client spawns a FilmRequest actor to search for the movie in servers. The FilmRequest actor sends messages to all of the servers and the first server which can provide the movie, spawns a Connection actor to start streaming. Besides, there are some indexer actors which are responsible for indexing the movies in the servers to make searching for movies easier.

In contrast to the aforementioned models, the load profile of actors in the FileService model may change during the time. This change takes place by requests migration when a server crashes. As soon as detecting a crashed server, requests which are sent to that server are distributed among other servers and the status of the crashed server is changed to *repairing*. Servers will back to service after passing a repairing period. The crash times of servers are generated by a Poisson distribution and we make sure that there is no case where all of the servers are in repairing state.

Preparing the cVector of actors of FileService for the Static-Heuristic approach results in mapping all actors to thread-pool policy, except the Client actor. However, the efficient mapping of the Server actors deeply depends on the load profile of the system. Assume that the actors are deployed as shown in Figure 8(b). In this configuration, having many film requests needs mapping Server actors to pinned to be able to process the request. This mapping reduces the performance of Indexer of that node but increase the performance of the system in general. By reducing the number of request, the mapping has to be changed to thread-pool to allow Indexer use more CPUs.

To illustrate the applicability of the Adaptive-Heuristic approach, we simulated the model in different configurations. In this case, instead of computing the



(a) The actor model of FilmService example

(b) The deployment diagram of the example

Fig. 8. The FilmService example

average completion time of tasks, we simulated the model for a long period of time and measure the utilization of CPUs using the Adaptive-Heuristic, Static-Heuristic, and default policy approaches. This way, as the needed computation power of all tasks are the same for the three approaches, the best policy has to fully utilize CPUs. So, better utilization of CPUs means completing more tasks in a given period of time.

As shown in Figure 9, the Adaptive-Heuristic approach is the only case that shows an acceptable behavior when the number of processors is increased. The figure also shows that the inefficiency of the Static-Adaptive approach is increased by increasing the number of CPUs; however, the Adaptive-Heuristic approach encounters a very small performance penalty.

We also examined the behavior of the model in the presence of many servers, depicted in 10. This figure shows that increasing the number of servers results in a light decrease in the performance of the system in the case of using the Static-Heuristic approach which is in contrast with the light increase which is depicted in the case of the default policy and the Adaptive-Heuristic approach.

5 Conclusion, Discussion and Future Works

In this paper, we proposed a new approach for associating threads to actors of a system. Applying the previously proposed approaches results in performance improvement; however, it relies on static analysis of source codes and actors' behaviors. In practice, relying on the static analysis of the codes and ignoring the runtime load profile of the application results in inappropriate configuration of systems in distributed environments. In contrast, the self-adaptive approach

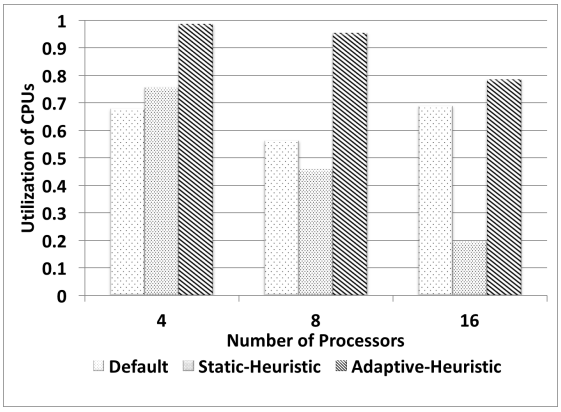


Fig. 9. Comparing CPU utilization in the case of having different number of CPUs

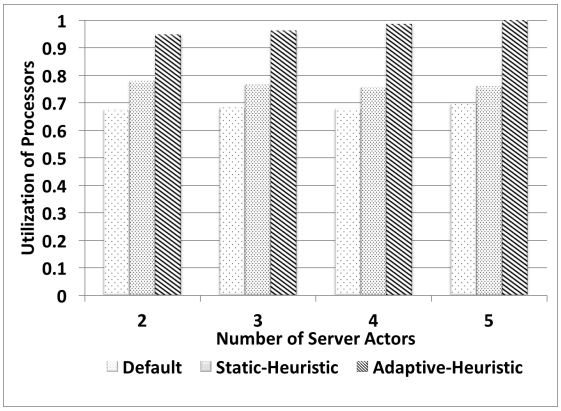


Fig. 10. Comparing CPU utilization in different configurations

tunes the mapping of the actors based on the captured information during the execution of the system. In this approach, the needed information can be gathered using very light-weight processes. Comparing the new approach with the old one using a set of case studies showed that the self-adaptive approach improves the performance of the systems in most cases.

Although we showed that the proposed approach results in performance improvements, as the results are computed using an actor simulation engine they may change in the real deployment of the actor models. So, we planned to develop the adaptation engine in Akka as the future work of this work. We also planned to develop more examples to show the effectiveness of the approach in different configurations.

Acknowledgments

The work on this paper has been supported in part by the project "Self-Adaptive Actors: SEADA" (nr. 163205-051) of the Icelandic Research Fund.

References

1. Agha, G.A.: ACTORS - A Model of Concurrent Computation in Distributed Systems. MIT Press series in artificial intelligence. MIT Press (1990)
2. Hewitt, C.: Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT (April 1972)
3. Lieberman, H.: Thinking about lots of things at once without getting confused: Parallelism in act i. Technical report, DTIC Document (1981)
4. Woelk, D.: Developing infosleuth agents using rosette: An actor based language. In: CIKM'95 Intelligent Information Agents Workshop. (1995) 1–2
5. Armstrong, J.: Erlang. Commun. ACM **53**(9) (2010) 68–75
6. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with salsa. ACM SIGPLAN Notices **36**(12) (2001) 20–34
7. Lightbend Inc.: Akka, http://akka.io
8. Francesquini, E., Goldman, A., Méhaut, J.: Actor scheduling for multicore hierarchical memory platforms. In Vinoski, S., Castro, L.M., eds.: Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop, Boston, Massachusetts, USA, September 28, 2013, ACM (2013) 51–62
9. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for java. In Vitek, J., ed.: ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings. Volume 5142 of Lecture Notes in Computer Science., Springer (2008) 104–128
10. Haller, P., Odersky, M.: Actors that unify threads and events. In Murphy, A.L., Vitek, J., eds.: Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings. Volume 4467 of Lecture Notes in Computer Science., Springer (2007) 171–190
11. Upadhyaya, G., Rajan, H.: An automatic actors to threads mapping technique for jvm-based actor frameworks. In Boix, E.G., Haller, P., Ricci, A., Varela, C., eds.: Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014, ACM (2014) 29–41
12. Rajan, H.: Capsule-oriented programming. In: ICSE (2), IEEE Computer Society (2015) 611–614
13. Vernon, V.: Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. 1st edn. Addison-Wesley Professional (2015)
14. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1) (2008) 107–113
15. White, T.: Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated). O'Reilly (2012)