NOP on multi-core architecture computers

Guilherme H. K. Martini, M. Sc.
Graduate Program in Electrical and Computer Engineering (CPGEI) - Student
Federal University of Technology – Paraná
Curitiba, Brazil
ghk.martini@gmail.com

Jean Marcelo Simão, Dr.
Graduate Program in Electrical and Computer Engineering (CPGEI) - Professor
Federal University of Technology – Paraná
Curitiba, Brazil
jeansimao@utfpr.edu.br

Robson Ribeiro Linhares, Dr.
Graduate Program in Electrical and Computer Engineering (CPGEI) - Professor
Federal University of Technology – Paraná
Curitiba, Brazil
linhares@utfpr.edu.br

***Abstract*—This short paper presents a comparison between different approaches to implement applications using the notification oriented paradigm on multi-core computers. Results indicate that many different paths can be used to achieve distribution of work with Akka.net being the most effective, productive and resourceful tool between the ones studied.**

Keywords—computing, multi-threading, computing efficiency, notification-oriented paradigm, multi-core, multi-agent systems.

I. INTRODUCTION

Modern computing deployment drives the need for the creation of new programming languages that would simplify and accelerate development of software. Doing more with less without losing control of what is being coded is one of the key reasons for continuous improvement of programming languages., This efficiency gain can mean either coding less and faster to solve a specific problem or it can mean that the same hardware architecture can become capable of doing more tasks simply by using a better suited programming paradigm [1].

The notification-oriented paradigm is a fairly new way of coding software systems that aims for better computer efficiency [2]. It has a naturally distributed system where methods, attributes and comparisons are stand-alone entities that notify each other, hence reducing the amount of wasted processing time used to poll unchanged data [2]. These characteristics make it a good and natural fit for architectures such as FPGA chips and manycore GPUs, but standard multi-core CPUs aren't an optimal target for it. Since the actual market consists of many x86 and x86-64 processors, as depicted on figure 1; and most of the high-performance modern computers in the world use this architecture, shown in figure 2, an approach for a multi-threading capability is proposed.

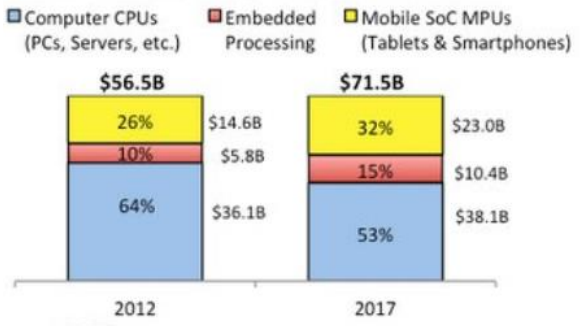

Fig. 1. Market share of x86-64 computers shift over time (light-blue), in dollars. Source: IC insights [3]

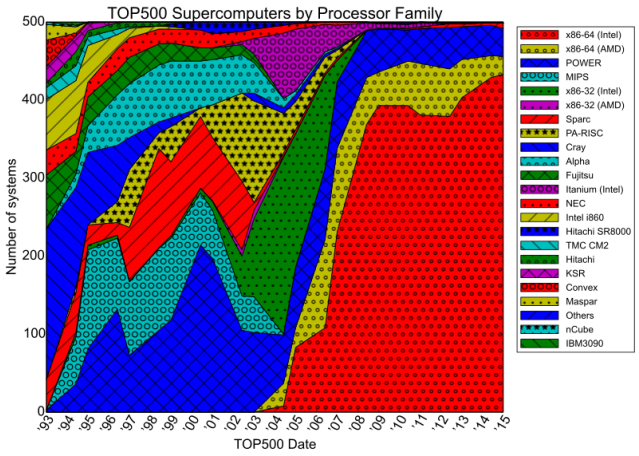

Fig. 2. Processor architectures of today's top 500 computers. Around 90% of them are x86-64. Source: Top500 [4]

II. BACKGROUND DEVELOPMENT AND OBJECTVES

The NOP language already has many different tools to enable its use such as: two compiler versions, an FPGA targeted compiler, a dedicated hardware architecture and many different applications that demonstrate its benefits and trade-offs. Also, a framework to enable the use of more than a single core of a multi-core computer is under development. This framework is intended to use C++ as the main technology to split the work of all NOP agents between all available cores by using Pthreads and by developing interlocks/mutexes to avoid racing conditions and deadlocks.

Some previous work on comparing a NOP application of an electronic gate system that made use of a simpler PThread approach with Thread Pooling approach was also done, and it proved that more refined parallel scheduling mechanisms can be a productive way to increase throughput. With that in mind, an initial study on how far actual technologies could be integrated to a NOP framework was necessary. Also, a study of how much new resources could be integrated to a NOP framework without compromising its performance became needed. So, the main goals for this work were sketched:

- To select a technology to aid on the multi-core development of NOP.
- To benchmark this technology with the C++ framework and with the simpler Pthread and Thread Pooling approaches.
- To clarify trade-offs and show the main advantages and disadvantages of each technology.

III.

III. Multi-core technologies Study

The first step of the work was to search for different technologies that could be used in order to fulfill some requirements, such as:

- To be able to support the NOP programming paradigm by being capable of supporting all features already implemented in this language.
- To be able to fully use a multi-core processor with a reasonable level of abstraction to the programmer, keeping the determinism of the applications.
- To be easy and productive to write code to it.
- To be easily expandable to a bigger program with a large number of entities.
- To be traceable and easy to debug.
- To be as expensive as the C++ framework in terms of processing cost, or, if possible, less expensive than it.

That lead to the research and reading of documentations of the following technology list:

- PThreads for C++
- Thread Pooling for C
- Erlang
- Haskell
- Node.js
- CSP
- Open CL
- Open MP
- FADALib
- Multi Agent Systems theory (MAS)
- Akka.net

There are many advantages and disadvantages in all technologies and in general all of them could have been studied and compared to the actual NOP frameworks. This work focused on comparing four different technologies. Two of them were already developed in a previous study, which are the PThreads for C++ and the Thread Pooling for C [5]. The third one is the already developed C++ NOP framework that needed some adjustments and also the coding of the same case of study, the electronic gate application. The fourth and last one would be a new technology and for this Akka.net was chosen. The list of reasons for choosing Akka.net follows:

- Developed over C#, Java or Scala
- Productive environment, Visual Studio
- Easy to debug and troubleshoot
- Well supported with wealth of online information
- Capable of abstracting threads and muxing
- Known to be of good performance
- Wealth of APIs for higher integration and expansion of study fields
- Scalable
- Distributable
- Fault-tolerant
- Based on the multi-agent theory

Between the ones that weren't chosen, Erlang has many similarities to Akka.net. The decision point between the two was the smaller productivity and support of Erlang when compared to Akka.net. Node.js and OpenMP are also noticeable in between the ones studied, but were left for a future study opportunity.

IV. Development in Akka.net

The first requirement to be able to work with Akka.net is to understand its actor model. Moreover, there is a need know the difference between an Actor Model and the NOP Paradigm. A simplified way to understand the difference is that the NOP paradigm does not encapsulate its entities as a service, so Premises, Conditions and FBEs exist logically, but not necessarily operate in a single sequential execution flow, meaning that function calls of FBE's methods can be stacked in different computer cores. In the Actor model, an Actor exist as an entity that enqueues its work, operate in a single context and then distribute message to other actors that can be running in other cores/contexts.

So, as a consequence, on a single computer, the NOP paradigm would be more distributable between cores than an agent system, but only when the sum of all work is less than all that is available. Whenever usage reaches 100%, that higher granularity won't make any difference. Also, a noticeable trade-off between the two is that the Agent system will be more suitable for distributed work due to its fault tolerance. Most likely the NOP paradigm will end up splitting work as actors would on a distributed environment so traceability of work is made on an easier way.

Akka.net coordinates its created actors under a K-ary three which separates the actors that are automatically created to coordinate the actor system to the ones that are created by the user, as seen in figure 1:

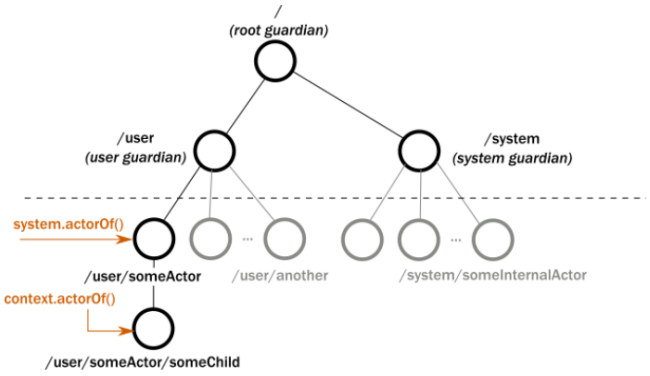

Fig.1. The Akka.net actor system structure

All actors created by the application would fall under the "/root/user" path and the ones needed to keep the actor system running are automatically created under the "/root/system" path. All actors are accessible via its http-like address, even under distributed systems. So, for the electronic gate application the actor system K-ary three would like what is depicted on figure 2.

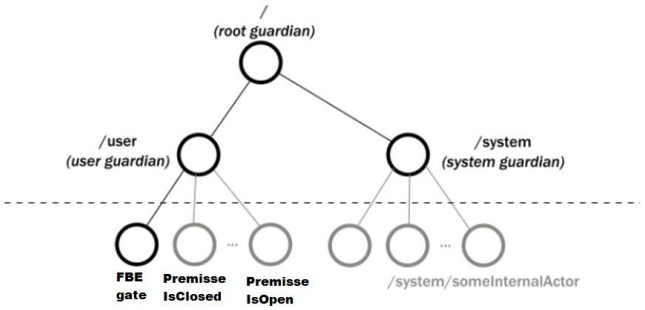

Fig.2. The Akka.net actor system structure
for the electronic gate application

So, the main three structure puts all NOP entities in the same three level and lets the main user guardian take care of all actors in case of faults.

It is worth mentioning that all actors communicate with each other by a message system: Every actor has a reference to the actors that need to receive their messages and all of them have mailboxes to enqueue incoming messages and take actions based on the message type and on the sender on the messages.

A deeper on-detail diagram of the actor system implemented for the electronic gate in Akka.net is depicted in figure 3, where which light blue container is an actor under "/root/user" path, and messages between actors were correlated to notifications from the NOP paradigm.

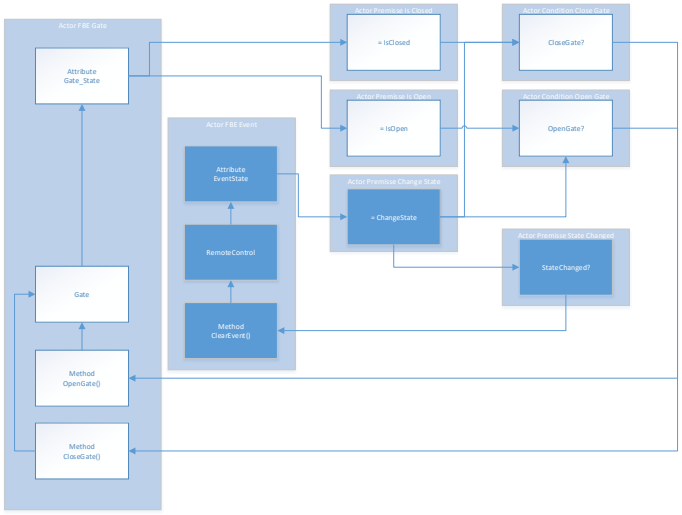

Fig 3. The electronic gate in the Akka.net actor model

As the first actor system model took shape, initial tests showed the same problem seen in the older thread pooling and PThread versions of the electronic gate: Event, methods, conditions and premises were being distributed between cores but without a minimum sequence that is necessary in order to keep determinism and coherence in the "sequence of facts" for a NOP program. As this implementation was made with the idea of keeping actors as standard as it can be, meaning that with little effort the system could run on a cluster or on a distributed network, a notification mechanism to keep all actions in sync and logically coherent was proposed, as seen in figure 4.

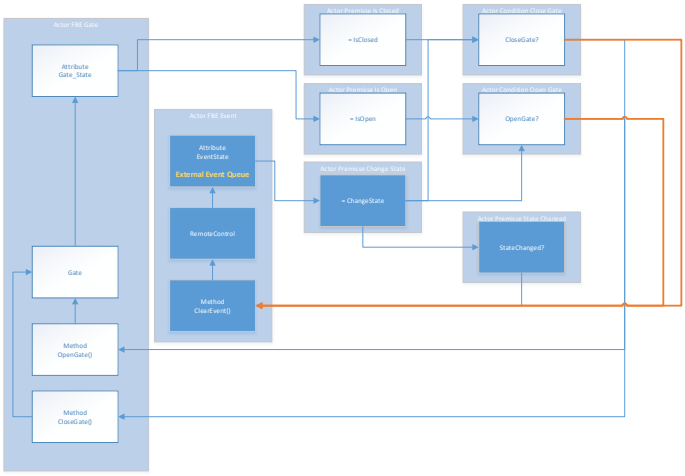

Fig 4. Akka.net logical interlock using notifications

This granted determinism and coherence on the application, having results similar to a single core-sequential application, but with the advantage of having each actor running on a different computer core. For future implementations, this would most likely become a NOP language feature where the developer will be in charge of knowing what has to be put in a logical sequence.

In order to make this feature work, two notifications and a queue of incoming external requests were implemented, being considered a relatively simple to develop approach.

A simple actor code is depicted in figure 5.

```
class ConditionStateChanged : UntypedActor
{
    private IActorRef _FBERemoteControlRef;


    private void SetFBERemoteControl(IActorRef FBERemoteControl)
    {
        _FBERemoteControlRef = FBERemoteControl;
    }

    protected override void OnReceive(object message)
    {
        if (message is ActorReference)
        {
            var temp = message as ActorReference;
            if (temp.ActorRefID == ActorRefType.FBERemoteControlRef) SetFBERemoteControl(temp.Ref);

            Sender.Tell(true);
        }
        else switch (message)
        {
            case ConditionAction.SendFalse:
                if (!_FBERemoteControlRef.IsNobody()) _FBERemoteControlRef.Tell(RemoteControlState.Off);
                Crc32_many_times();
                break;
        }
    }
}
```

Fig 5. Akka.net actor code

It can be seen that every actor is inherited from a base class called "Untyped Actor". Also, the OnReceive method must be overridden in order for the code to compile. In that method all message handling from the actor Inbox has to happen. For the NOP implementation, OnReceive is used to receive references and link actors, send notifications, trigger methods and turn conditions true/false. All references to send notifications are saved in private objects names IActorRef.

Creation of actors is pretty simple and look like a table as shown in figure 6, the last parameter passed inside the constructor is the actor name in the K-ary three.

```
//Build up all actors
IActorRef FBEGateActor             = NOPActorSystem.ActorOf(Props.Create(() => new FBEGate()),             "FBEGateActor");
IActorRef FBERemoteControlActor    = NOPActorSystem.ActorOf(Props.Create(() => new FBERemoteControl()),    "FBERemoteControlActor");
IActorRef PremisseIsClosedActor    = NOPActorSystem.ActorOf(Props.Create(() => new PremisseIsClosed()),    "PremisseIsClosedActor");
IActorRef PremisseIsOpenActor      = NOPActorSystem.ActorOf(Props.Create(() => new PremisseIsOpen()),      "PremisseIsOpenActor");
IActorRef PremisseChangeStateActor = NOPActorSystem.ActorOf(Props.Create(() => new PremisseChangeState()), "PremisseChangeStateActor");
IActorRef ConditionCloseGateActor  = NOPActorSystem.ActorOf(Props.Create(() => new ConditionCloseGate()),  "ConditionCloseGateActor");
IActorRef ConditionOpenGateActor   = NOPActorSystem.ActorOf(Props.Create(() => new ConditionOpenGate()),   "ConditionOpenGateActor");
IActorRef ConditionStateChangedActor = NOPActorSystem.ActorOf(Props.Create(() => new ConditionStateChanged()), "ConditionStateChangedActor");
```

Fig 6. Creating actors in Akka.net

Linking actors is also simple. After all are created, a sequence of messages (or notifications) send all references to all actors in the system, so the NOP notification chain is closed as seen in figures 3 and 4. Figure 7 shows how this is coded.

```
//Set all actor references
var FBEGateActorTask1             = FBEGateActor.Ask(new ActorReference(ActorRefType.PremisseIsClosedRef, PremisseIsClosedActor));
var FBEGateActorTask2             = FBEGateActor.Ask(new ActorReference(ActorRefType.PremisseIsOpenRef, PremisseIsOpenActor));
var FBERemoteControlActorTask1    = FBERemoteControlActor.Ask(new ActorReference(ActorRefType.PremisseChangeStateRef, PremisseChangeStateActor));
var PremisseIsClosedActorTask1    = PremisseIsClosedActor.Ask(new ActorReference(ActorRefType.ConditionCloseGateRef, ConditionCloseGateActor));
var PremisseIsOpenActorTask1      = PremisseIsOpenActor.Ask(new ActorReference(ActorRefType.ConditionOpenGateRef, ConditionOpenGateActor));
var PremisseChangeStateActorTask1 = PremisseChangeStateActor.Ask(new ActorReference(ActorRefType.ConditionCloseGateRef, ConditionCloseGateActor));
var PremisseChangeStateActorTask2 = PremisseChangeStateActor.Ask(new ActorReference(ActorRefType.ConditionOpenGateRef, ConditionOpenGateActor));
var PremisseChangeStateActorTask3 = PremisseChangeStateActor.Ask(new ActorReference(ActorRefType.ConditionStateChangedRef, ConditionStateChangedActor));
var ConditionCloseGateActorTask1  = ConditionCloseGateActor.Ask(new ActorReference(ActorRefType.FBEGateRef, FBEGateActor));
var ConditionCloseGateActorTask2  = ConditionCloseGateActor.Ask(new ActorReference(ActorRefType.FBERemoteControlRef, FBERemoteControlActor));
var ConditionOpenGateActorTask1   = ConditionOpenGateActor.Ask(new ActorReference(ActorRefType.FBEGateRef, FBEGateActor));
var ConditionOpenGateActorTask2   = ConditionOpenGateActor.Ask(new ActorReference(ActorRefType.FBERemoteControlRef, FBERemoteControlActor));
var ConditionStateChangedActorTask1 = ConditionStateChangedActor.Ask(new ActorReference(ActorRefType.FBERemoteControlRef, FBERemoteControlActor));
```

Fig 6. Linking actors in Akka.net

This finishes the development phase for the Akka.net, being considered a truly productive environment, well supported and straightforward for the NOP paradigm.

V. Development with the C++ Framework

The development work with the framework consisted firstly in removing the bulk of the framework in order to make it run a simple application as the electronic gate. Some performance measurements, text dumping and some unused data structures were removed. Later, the electronic gate was developed and the NOP entities were tied up, as seen in figure 7.

```
CoreControllersManager::createCoreControllers(8);

//Gate notification cycle
Entity *attribute1 = new Entity("gate_state");
attribute1->setCore(0);

Entity *premise1 = new Entity("premise_is_closed");
premise1->setCore(1);
attribute1->registerEntity(premise1);

Entity *premise2 = new Entity("premise_is_open");
premise2->setCore(2);
attribute1->registerEntity(premise2);

Entity *condition1 = new Entity("condition_close_gate");
condition1->setCore(3);
premise1->registerEntity(condition1);
```

Fig 7. Linking actors in the C++ framework

Then, a deeper study on how actions/events were sent to different cores in the computer were made, and that showed some limitations in the framework where all events/actions related to and Entity (that can be an FBE, condition, premise, etc) are directed to the same computer core since it is the Entity what holds the information of where it must be run and this isn't mutable along the execution.

Figure 8 shows an Entity, which contains a list to notify all other actors, but it lacks support on abstracting objects and sending them as parameters of the notifications, which is a native resource on Akka.net.

```
void Entity::setCore(short id) {
    coreId = id;
}

void Entity::registerEntity(Entity *entity) {
    entities.addLastElement(entity);
}

void Entity::onNotification() {
    CoreControllersManager::registerNotifier(coreId, this);
}

void Entity::execute() {

    //std::cout << "Executing " << name << " ["<< coreId << "]" << std::endl;

    entities.initIterator();

    while (entities.hasNext()) {
        entities.next()->onNotification();
    }

    //Method function calls

    exec_count++;

}
```

Fig 8. Basic actor in the C++ framework

VI. Performance Comparisson

The performance check between all four versions was made by a simple clock counter. Whenever the code execution started, the clock counter starts. When the application is finished, the clock count is summed up and translated to an estimated time that was taken by the computer to run the application. All runs were made under the same computer, with the same operational system and with the same task priority on its scheduler.

As the Electronic gate project is a simple program, a CRC32 calculation was added to every method that is called in order to increase processing burden. To further increase that burden and to evaluate how more costly programs would run, a "for" loop was added to the CRC32 calculation, so a tendency can be verified as burden is changed.

Many runs for the same amount of CRC32 loops were made in order to confirm that timing was consistent between them. For a same simulation, times varied less than 1% in all attempts. Since PThread and Thread Pooling were tested in dividing work between two cores, two tests were done for the C++ Framework and for the Akka.net versions: One splitting the work only in two cores also, and another dividing the same workload between all eight cores of the same computer (called "even" versions).

Figure 9 shows a graph with the results. As processing burden increases, the cost of setting up an actor system becomes negligible, and a lot of gains comes with it: data integrity, abstraction from threads, no deadlocks, notifications passing abstract objects, scalability, etc. Most of these capabilities are not natively present in the ThreadPooling version or in the PThread version.

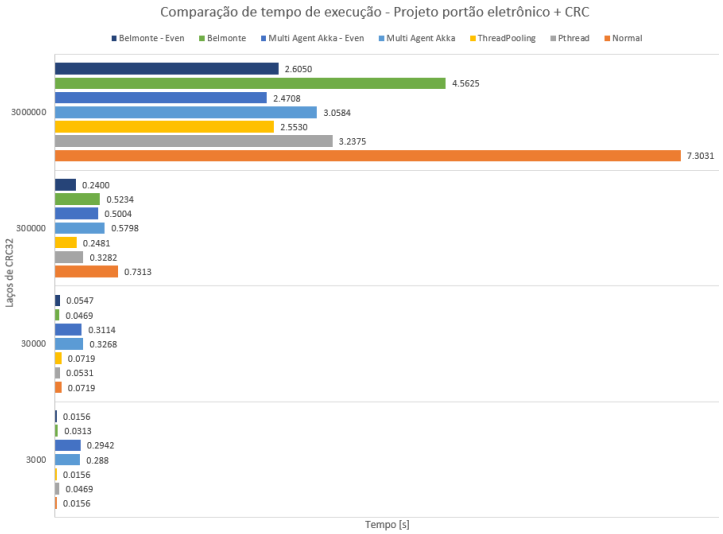

Fig 9. Tests results

A second test was made to deeply compare the C++ Framework and the Akka.net version. In this comparison, 3000 CRC32 loops were calculated in each Entity/Actor whenever a notification was received. Every 3 actors were tied to each other on a ring of notifications. At the beginning of the application a message would trigger one of the actors of the ring that will consequently trigger the other two. The difference in this test is that the burden is kept constant while the number of actors were increased linearly. Results are plotted on figure 10.

Some issues were found on the C++ framework were many of the data structures are dynamically allocated, making use of the heap memory. Even when creating the Entities statically, its internal functions led to a sequence of heap allocations that didn't allow testing past 7200 actors on a single simulation.

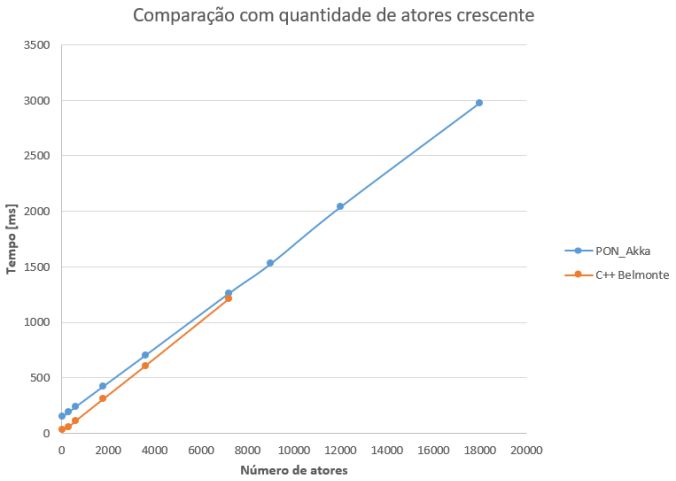

Fig 10. C++ framework and Akka.net comparison
With increasing number of actors

As the graph shows, initially the Akka.net doesn't perform as well as the C++ framework but this difference is then compensated when the number of actors increase.

VII. Conclusion

With the gathered data it is concluded that all goals for this work were achieved, and that any differences between the C++ framework and the Akka.net are nearly negligible if not favorable to the Akka.net. Previous implementations showed good numbers but lack in tooling and resources for futures applications, and their implementation might lead to the same performance that Akka.net already has. Akka.net is a path that can take the NOP paradigm to a better productivity level, speeding up scientific research on distributed, multi-core and fault tolerant systems.

References

[1] Rojas, Raúl; et al. *Plankalkül: The First High-Level Programming Language and its Implementation*. Institut frame Informatik, Freie Universität Berlin, Technical Report B-3/2000, February, 2000. Avaliable: ftp://ftp.mi.fu-berlin.de/pub/reports/TR-B-00-03.pdf

[2] Banaszewski, Roni; et al. *Paradigma Orientado a Notificações: Avanços e Comparações*. UTPFR, 2009.

[3] *The McClean report [Online]*. IC Insights, 2018.
Avaliable:http://www.icinsights.com/services/mcclean-report/report-contents/

[4] *Statistics | TOP500 Supercomputer Sites [Online]*. Top500, 2014.
Avaliable: https://www.top500.org/statistics/

[5] Seferidis, Johan Hanseen. *C-Thread-Pool [Online]*, April, 2017.
Available: https://github.com/Pithikos/C-Thread-Pool