



BIROn - Birkbeck Institutional Research Online

Mannock, Keith and Aning, Kwabena (2017) An architecture and implementation of the actor model of concurrency. In: The 8th International Conference on Information, Intelligence Systems and Applications, 28 – 30 Aug 2017, Larnaca, Cyprus. (Unpublished)

Downloaded from: <https://eprints.bbk.ac.uk/id/eprint/20836/>

Usage Guidelines:

Please refer to usage guidelines at <https://eprints.bbk.ac.uk/policies.html>
contact lib-eprints@bbk.ac.uk.

or alternatively

An Architecture and Implementation of the Actor Model of Concurrency

Kwabena Aning

Department of Computer Science and
Information Systems
Birkbeck College, University of London
London, WC1E 7HX
Email: k.aning@dcs.bbk.ac.uk

Keith Leonard Mannoek

Department of Computer Science and
Information Systems
Birkbeck College, University of London
London, WC1E 7HX
Email: keith@dcs.bbk.ac.uk
Contact author

Keywords—Distributed computing, Design and implementation, Software Architectures, Parallel and Concurrent Computing, Agent Architecture, Programming languages.

Abstract—In this paper we describe an architecture and implementation of the ACTOR model of concurrent computation which exploits the multi-core processors of modern day computer architectures. A novel aspect of our approach, and where it differs from many other implementations, is that it is hosted in an existing programming language as native constructs; we employ Swift which is rapidly rising in popularity but in its standard distribution lacks the facilities for true concurrent programming. We describe an extension to the language which enables access to concurrent features and provides an API for supporting such interactions. We consider the various architectural issues, competing approaches, and discuss early findings from our prototype implementation.

I. INTRODUCTION

As we have come nearer to the limitations of current processor technology there has been a growing movement towards the use of processors with several cores. The aim of these processor architectures is to improve the throughput, efficiency, and processing power of the computer but these benefits do not come without their own problems and challenges. One major hurdle with this approach is building software that can leverage these facilities while still being a tractable programming model. This can lead to quite low-level constructs which utilise shared resources, to enable the promised processing power with multi-core computing.

In the Section II, we will outline the differences between concurrency and parallelism; there is often confusion between these terms so we need to clarify these. In Section III we outline the rationale for our work and describe the background work and related systems that underpin this work. This includes descriptions of alternative concurrency models and other aspects relevant to our work. In Section IV of this paper we describe our architecture for our Actor implementation that addresses some of the problems with concurrency, namely non-determinism, deadlocking, and divergence, which leads to a more general problem of shared data in a concurrent environment. We also briefly describe our implementation and our message passing strategy. We conclude the paper with a discussion of future work.

II. CONCURRENCY AND PARALLELISM

To provide a context to the sections that follow, a distinction needs to be made between concurrency and parallelism. Concurrent programs are best described as an interleaving of sequential programmes[1]. A concurrent program has multiple logical threads of control. These threads may or may not run in parallel [2]. Tony Hoare expresses this concept of concurrency as follows:

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q \quad (1)$$

Assuming that P is a process that involves writing logs to a given file from a given input. Q is a process that also involves reading that given file and displaying its contents to a given output. $\alpha(P \parallel Q)$ is therefore all the behaviours that are involved with writing logs to a file, reading logs from a file, and displaying those logs to some output. If our given environment allows for it, any these behaviours can occur at any time, in any sequence. The above equation describes a coming together of these processes where any of these behaviours can be called upon with no contingency.

”A process is defined by describing the whole range of its potential behaviour.“ [3]. In other words, to define a process completely one has to enumerate all the potential behaviours or properties of that process. These can be referred to as in CSP *alphabets*. In CSP alphabets are usually denoted by α can be described as all the set of names, behaviours, and/or actions that are considered relevant for a particular description of an object or in this case a process.

Given the events in the alphabets of processes αP and αQ respectively, which requires simultaneous execution, P can participate in any or all of its alphabets without affecting or concerning Q and vice-versa, and as such all events are logically possible for this system — a union of all alphabets [3].

It is also important to stress that there is an *order* element to the definition of concurrency. In that tasks can be performed in any order, and this allows for parallelism as the tasks can then be shared between several processes if the order that they are performed does not matter.

Parallelism may be seen as an latent benefit of concurrently written programs. A parallel program is one whose tasks can be distributed across more than one process. This does not imply that the program is working on different tasks at once. It simply implies that the program is written in such a way so that different parts of it, or its computations can be run or can be performed on different processors simultaneously. Using the illustration above, writing the logs to the file could be executed on one processor, while reading the file could be done on another. In all cases, this is possible because processes P and Q can run independently of each other.

III. RATIONALE, BACKGROUND, AND RELATED WORK

When more than one process requires access to a shared resource one has to make certain decisions about the granularity of access, where the coarser the access the less concurrency that is available (in general). The accepted wisdom in this area is to either incorporate features into the programming language, e.g., threads in Java, or leverage a *toolkit* or library. These low level constructs can lead to extremely complex interactions between the various processes leading to contention, with the possibility of deadlock, race-conditions, etc. Some programming languages have focussed on these problems, e.g., Erlang, but they haven't transitioned to the mainstream. Other programming languages have also taken these issues on, e.g., *Pony* [4], *Go* [5], *Scala* [6], and *Clojure* [7], with varying levels of success.

We adopted the ACTOR model of concurrency [8] to address these problems. We considered making this available as a library or toolkit but decided the Java approach, of building constructs into the base language, to be a better approach. We didn't want to fall into the trap of making the constructs too low level, or to make them, and the model, difficult for the programmer to comprehend and use. We decided to enhance the Swift programming language as:

- it is open-source,
- it is efficient,
- it isn't *yet another programming language* which hardly anyone will use,
- it is compiled,
- it is available on several platforms (outside of the Apple world),
- it is an object-oriented language with functional programming paradigm features, and
- it is growing rapidly in popularity.

By adopting a mainstream language with less *baggage* than existing languages we get the best of multiple worlds; a potentially large user community, and a reliable and efficient programming language. The success of ACTOR frameworks such as Akka for Scala and Java suggested a way forward for Swift that would add to the functionality of the language while addressing concurrent processing issues.

A. Related Work

As stated previously our work builds upon the ACTOR model but we also owe some concepts and motivation to a

number of other technologies and programming languages. For the remainder of this section we describe these features.

B. The Akka Library

Earlier versions of Scala had natively implemented actors as part of the Scala library and could be defined without any additional libraries. Newer versions (2.9 and above) have removed the built in *Actor* and now there is the Akka Library.

Akka is developed and maintained by TYPESAFE [9] and when included in an application, concurrency can be achieved. Actors are defined as classes that include or *extend* the *Actor* trait. This feature enforces the definition of at least a *receive* function. *Receive* is defined as a partial function, taking another function and returning a `Unit` (void value).

The function it expects is the behaviour that the developer needs to program into the actor. This is essentially defined as a pattern matching sequence of actions to be taken when a message is received that matches a given pattern.

```
import akka.actor.Actor
class MyActor extends Actor {
  def receive = {
    case Message1 =>
      //some action
    case Message2(x: Int) =>
      // another action use
      // x as in int
    ...
    case MessageN =>
      //Other actions
  }
}
```

At the heart of the Akka Actor implementation is the Java concurrency library `java.util.concurrent` [10]. This library provides the *(multi)threading* that Akka Actors use for concurrency. Users of the library do not need to worry about scheduling, forking and/or joining. This is dealt with by the library's interaction with the executor service and context.

The Akka Library offers options to select which *executor service* to use. It currently defaults to the *ForkJoinPool*, which is usually sufficient for most tasks. This is referred to as the *Dispatcher* [9][11] and is equipped with the functionality to determine the execution strategy for a given program, such as which thread to use, how many to make available in a pool for actors to run on, etc.

All these options are exposed in a malleable *configuration factory*. Developers are able to fine tune the actor system's behaviour which relative ease.

C. Kotlin coroutines

Kotlin, a language developed by JetBrains, has a feature which implements the known technique of *coroutines*; these can best be described as operations that can be suspended and resumed at a later time, potentially using a different thread of execution.

D. Concurrent ML

Concurrent ML is described as a functional concurrent language [12] designed for high performance. Standard ML, a statically typed programming language with an extensible type system, which supports both imperative and functional programming paradigms, was extended to incorporate concurrent formalisms, most notably, synchronous message passing, and is similar to CSP with the messages being passed over typed channels [13].

It extends Standard ML with

```
val spawn : (unit → unit) → thread_id
type 'a chan
val channel : thread_id → 'a chan
val recv : 'a chan → 'a
val send : ('a chan * 'a) → thread_id
```

A CML program runs by spawning processes called threads that pass messages over typed channels. So that only a certain data type can be passed over that channel to another process with a channel that accepts that same data type. This extension provides primitives for implementing concurrency directly in ML. They are referred to as threads here to make a clear distinction between operating system processes and those spawned in CML; threads in CML are lightweight.

As per the listing above, a CML program begins by spawning a single thread which is often the control thread — the *parent* thread, which can in turn spawn other threads. The *spawn* function takes another function as an argument and creates a thread for it by using *call by value*. Execution is delayed until that value is required, with the thread created at the time its needed. This new thread referred to as the *child*, will be terminated once the execution of that function for which it was spawned is completed. CML keeps the *parent* and *child* threads discrete and as such a terminating *parent* thread does not cascade to its children. A thread may also be terminated by calling an exit function.

```
val exit : unit → 'a (2)
```

This can be equated to an exception thrown in other programming languages where its result type is 'a as it never returns. Also a thread may terminate if the executing process, raises an exception with no handler defined for it. By default this exception will not reach its parent although there are means of reporting this upstream.

The CML runtime also multiplexes the processors for the threads produced by a CML program. Given that the number of potential threads spawned is unbound[1], this step is necessary in managing the processor resources. Threads can be used liberally in CML because they are represented by SML values making them very lightweight in terms of space overhead, and can also be garbage collected. Concurrent ML provides mechanisms for communication between these multiple threads. Message passing can be either synchronous or asynchronous over typed channels.

```
type 'a chan (3)
```

Is a type constructor used to generate typed channels. Two operations are natively provided for communicating on this type to a generated channel.

```
val recv : 'a chan → 'a
val send : ('a * 'a) → unit (4)
```

Channels can be viewed as meeting or *rendezvous* points where processes simply communicate. Channels do not specify direction of target a particular process, they are just points where messages are sent and received.

selective communication — a key mechanism in message-passing concurrent languages [1] allows threads to block on non-deterministic choice of several blocking communications. Using the *select* keyword, we are able to choose between two or more events that are simultaneously enabled non-deterministically. “*Select* is syntactic sugar for the composition of the *sync* operator and the *choose* event combinator[1]” The *sync* operator forces a synchronisation on an event value and the “*choose* combinator provides a generalised selective communication mechanism”[12]. It takes a list of events and returns an event that represents the non-deterministic choice of events[1].

E. Occam 2

Occam is an imperative procedural language, implemented as the native programming language for the Occam model. This model also formed the bases for the hardware chip - the *INMOS transputer microprocessor* [14]. It is one of several parallel programming language developed based on Hoare’s CSP[3]. Although the language is a high level one, it can be viewed as an assembly language for the transputer [14]. The transputer was built with four serial bi-directional links to other transputers providing message passing capabilities among the transputers. Concurrency in Occam is achieved by message passing along point-to-point channels, that is, the source and destination of a channel must be on the same concurrent process.

```
VARIABLE := EXPRESSION
CHANNEL ? VARIABLE
CHANNEL ! VARIABLE
```

This notation takes its exact meaning from Hoare’s CSP [3]. The “?” is requesting an input from the channel to be stored in the VARIABLE whereas “!” is sending a message over the channel and the message is the value stored in VARIABLE. Occam is A strongly typed language and as such the channels over which messages are passed need to be type safe. The type can be *ANY* meaning that the channel can allow any type of data to be transmitted over it. An inherent limitation in Occam’s data structures are that the only complex data type available is the *ARRAY*.

The language enables several processes be executed in parallel using the *PAR* construct.

```
PAR
INT x:
chan1 ? x
```

```
INT y:
chan2 [?] y
```

Any number of processes can be put into the `PAR` construct and it only terminates when all the component processes have terminated, either successfully or not.

Also worth mentioning is the `ALT` construct which implements Dijkstra's guards on processes. When an `ALT` guard is used, execution will wait until at least one of the branches are satisfied. Satisfaction could include boolean results, channels being ready, or timers.

There are other constructs available to Occam programmers but the above are the simple ones that are specific to concurrency in Occam and in the spirit of Occam's razor, form a large part of the programming language's philosophy.

F. Erlang

The Erlang Virtual Machine provides concurrency for the language, in a portable manner and as such it does not rely to any extent on threading provided by the operating system nor any external libraries. The self contained nature of the virtual machine ensures that any concurrent programmes written in Erlang run consistently across all operating systems and environments.

The simplest unit in the language is a lightweight virtual machine called a *process* [15]. Processes communicate with each other through *message passing*. A simple process written to communicate between processes could be:

```
start() -> spawn(module_name, [Parameters])
loop() ->
receive
  pattern -> expression;
  pattern -> expression;
  pattern...n -> expression;
end
loop().
```

`start()` spawns the process for the current module with any parameters that are required. A loop is then defined which contains directives to execute when it receives messages of the enumerated patterns that follow — `loop()` is then called so that the process can, once again, wait to receive another message for processing.

The above will code fragment will exhibit the behaviour pattern below:

$$S * E \rightarrow A, St \quad (5)$$

Therefore, given a *state* S with an occurrence of an *event* E , some *action(s)* A should be performed that transitions our process to a new *state* St . In this case *expression* is the representation of the transition of the program from one state to the other.

a) *Erlang/OTP*: Erlang in itself provides several constructs for writing concurrent programmes in ways that allows for runtime optimisation and fault tolerance. Capabilities such as *hot code swapping*, *links*, *monitors*, *supervisors*, *timeouts* and so on are all built in capabilities available to an Erlang programmer. These have to be coordinated manually to achieve

the expected result. The OTP framework provides a library for grouping these error-prone manual processes into well tested and coordinated best practises and standards. OTP does most of the heavy lifting for the developer interested in concurrency and also providing minimalistic boilerplate code for required behaviours of generic applications. The OTP library is distributed with all modern versions of the language environment and as such can be considered as part of the Erlang standard distribution. Behaviours, or *interfaces*, are defined, but it is up to the developer to provide the business logic within the applications that use any generic behaviours. The OTP library also provides best practices for structuring Erlang code.

G. Pony

Pony is an object-oriented, actor-model, capabilities-secure programming language [4]. In object oriented fashion, an actor designated with the keyword *actor* is similar to a class except that it has what it defines as *behaviours*. Behaviours are defined as asynchronous methods defined in a class. Using the *be* keyword, a behaviour is defined to be executed at an indeterminate time in the future[4].

```
actor AnActor
  be(x: U64) =>
    x * x
```

Pony runs its own scheduler using all the cores present on the host computer for threads, and several behaviours can be executed at the same time on any of the threads/cores at any given time, giving it concurrent capabilities. It can also be viewed within a sequential context also as the actors themselves are sequential. Each actor executes one behaviour at a given time.

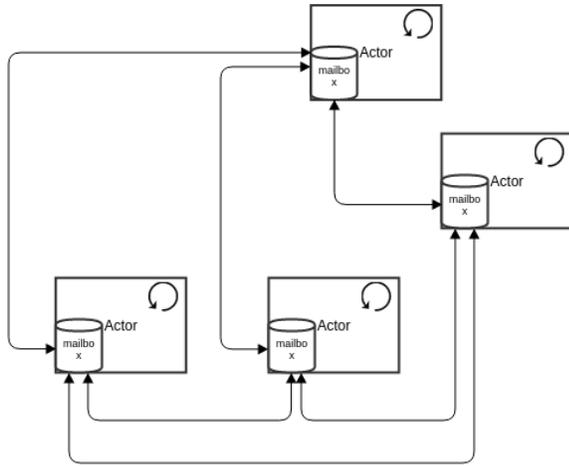
IV. ARCHITECTURE

Our architecture is based upon the fundamental constructs of the Actor model; its *axioms*. All actor based models need to exhibit the following properties:

- 1) Encapsulation,
- 2) Internal State,
- 3) Messaging,
- 4) Indeterminacy, and
- 5) Mobility.

Encapsulation is well understood and therefore we won't elaborate upon it further. *Internal State* follows on from encapsulation but really enforces the access *firewall* around the item; only the actor itself can access its internal state. *Messaging* encompasses asynchronous and synchronous message passing mechanisms. This also ties up with the final point, *Mobility*, as the actor should exhibit location transparency, an essential characteristic of a fully modular and versatile computational environment. The remaining point, *Indeterminacy*, means that there is no order to the messages received, or at least that this is not guaranteed. Messages may arrive in any order and it is the responsibility of the actor to process those messages in a consistent manner.

The following figure illustrates actors interacting with each other — each of which autonomously runs in it’s own process signified by the circular arrow. Each actor has there own attached mailbox, to which each of the actors can send messages.



Basic actors

Therefore each individual actor has:

- An internal state, which is only mutable by itself.
- A mailbox into which it receives messages.
- An internally accessible method for interacting with those messages.
- An implementation of a protocol that allows it to communicate with other actors.

There are several components to our architectural model which we will now (briefly) describe together with their implementation.

A restriction is that an actor can only be created within a context to ensure that the actor has all the necessary properties to communicate with other actors (within the current context). As this work is embedded in the Swift programming language, the *Grand Central Dispatch* library [16] is utilised for the MacOS implementation. We also have an alternative implementation for the open source version of the language which runs on other platforms; this implementation is not as advanced in its development. Therefore we concentrate on the MacOS implementation for the description included in this paper.

A. The Actor Context

This is the logical domain for the creating and running of actors. The context defines a *namespace* for actors, allowing actors within that same context to communicate with each other. The context is also responsible for creating the actors as they cannot be and should not be instantiated in isolation.

An extract from the *interface* is shown below.

```
protocol ActorContextProtocol {
    var contextName: String {get set}
```

```
var childActors: [String: MailBox]
    {get set}
var orphanedMailboxes: [MailBox]
    {get set}
func mailboxes() -> [MailBox]
func children() -> [String: MailBox]
func addChild(actor: ActorProtocol,
    actorMailbox: MailBox)
func removeChild(actor: ActorProtocol)
}
```

As can be deduced from the protocol defined above the context is responsible for keeping track of all actors created within it, and it also keeps track of the mailboxes assigned to actors. Once actors have been removed from that context the orphaned mailboxes can be reassigned to actors that are created to replace them.

B. The Actor

This is the main unit of computation. The actor defines the means to process messages it receives and any other business logic associated with the work that it does. The actor is attached to but does not own a mailbox from which it receives its messages. It is to be noted that it is within the actor that the a different thread is spawned for the work to take place. It also worthy of note that the process of spawning new threads should be independent of the actor and as such interchangeable. The actor implementation within swift will be done in a *type safe* manner so that message types are defined. This has the added advantage of predictable message handling on the side of the Actor.

```
protocol ActorProtocol {
    var shuttingDown: Bool {get set}
    var actorContext: ActorContext {get set}
    var identifier: String {get set}
    init(name: String, context: ActorContext)
    func tell(msg: NSObject,
        sender: ActorProtocol?)
    func processor(msg: (message: NSObject,
        sender: ActorProtocol?)) -> Void
}
```

The actor has methods for “telling”, or sending messages to other actors. The `processor` method takes a message and returns nothing. This is the component of the actor that does the work. The processor is invoked on a separate thread using the *Dispatch Framework* [16]. The actor will continuously *poll* the mailbox attached to it to ascertain whether it has messages or not. If it does it invokes the defined processor on a separate thread with the message it has just received from its mailbox.

C. Mailbox

The Mailbox may also be referred to as a message queue attached to at least a single actor. This is where messages are sent to. So that the actors never directly receive messages. The *Actor Context* is responsible for routing messages to the given actors mailbox and the actor then picks up the message from its attached mailbox. This is to ensure that should an actor stop

working for any reason such as entering into an exceptional state and receiving a termination message from its supervisor, messages that are sent to that actor while it is shutting down are not lost and as such *buffered* in the mailbox, waiting for the next actor to take control.

This component is implemented as a simple typed queue. A collection that accepts and provides an API for storing and retrieving homogeneous messages.

```

struct MailBox: Hashable {
    var items = [(msg: NSObject,
                  sender: ActorProtocol?)]()
    mutating func push(_ item: NSObject,
                       sender: ActorProtocol?)
    mutating func pop() -> (NSObject,
                           ActorProtocol?)
    func isEmpty() -> Bool
    var hashValue: Int
    static func ==(lhs: MailBox,
                  rhs: MailBox) -> Bool
}

```

V. CONCLUSIONS AND FUTURE WORK

In this paper we have briefly outlined the rationale for an Actor based model of computation to enable better utilisation of current processor architectures (namely, multi-core). We then briefly described the architecture and implementation of our prototype *system* which has been embedded into the popular Swift programming language. Our prototype has the functionality to enable a programmer to write actor based programs, without the complexity of low-level concurrent features, and is per-formant with hand-crafted concurrent code.

We plan to continue developing the functionality of the model and we are working on the model and implementation to improve its performance in terms of memory utilisation and ease-of-use. We are extending the architecture to allow the interoperability of discrete actor systems, enabling our actor model to function efficiently in cloud-based environments. All of this will enhance the flexibility and performance of our actors.

At present the model is difficult to extend across machines because we do not have a common message bus so we also plan to add an *event bus* which will add further scheduling facilities, logging mechanisms, and further monitoring capabilities, which will be able to monitor the various actor systems and their interactions. We will extend the model with further testing, examples, and various benchmarking measures.

The current implementation is limited to OSX but as Swift is available in open-source, and cross-platform, we are actively porting our model to additional platforms. To further show the extensibility and flexibility of our model we are also considering hosting its functionality in another programming language, e.g., Kotlin, which is JVM based and native, providing us with experience of our implementation outside of LLVM based systems[17].

REFERENCES

- [1] J. Reppy, *Concurrent Programming in ML*. Cambridge University Press, 2007. [Online]. Available: https://books.google.co.uk/books?id=V_0CCK8wcJUC
- [2] P. Butcher, *Seven Concurrency Models in Seven Weeks: When Threads Unravel*, 1st ed. Pragmatic Bookshelf, 2014.
- [3] C. A. R. Hoare, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.
- [4] S. Clebsch. (2015, 01) The pony programming language. The Pony Developers. [Online]. Available: <http://www.ponylang.org/>
- [5] R. Pike, "Go at google," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12. New York, NY, USA: ACM, 2012, pp. 5–6. [Online]. Available: <http://doi.acm.org/10.1145/2384716.2384720>
- [6] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, 2nd ed. Artima Inc, 2011.
- [7] R. Hickey, "The clojure programming language," in *Proceedings of the 2008 Symposium on Dynamic Languages*, ser. DLS '08. New York, NY, USA: ACM, 2008, pp. 1:1–1:1. [Online]. Available: <http://doi.acm.org/10.1145/1408681.1408682>
- [8] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1624775.1624804>
- [9] Typesafe. (2014, Feb.) Build powerful concurrent & distributed applications more easily. [Online; accessed 05 Feb, 2014]. [Online]. Available: <http://www.akka.io/>
- [10] D. Wyatt, *Akka concurrency*. Artima Incorporation, 2013.
- [11] T. Lockney and R. Tay, "Developing an akka edge," 2014.
- [12] J. H. Reppy, "Cml: A higher concurrent language," *SIGPLAN Not.*, vol. 26, no. 6, pp. 293–305, May 1991. [Online]. Available: <http://doi.acm.org/10.1145/113446.113470>
- [13] J. Reppy, C. V. Russo, and Y. Xiao, "Parallel concurrent ml," *SIGPLAN Not.*, vol. 44, no. 9, pp. 257–268, Aug. 2009. [Online]. Available: <http://doi.acm.org/10.1145/1631687.1596588>
- [14] D. C. Hyde, "Introduction to the programming language occam," 1995.
- [15] J. Armstrong, *Programming Erlang*. Pragmatic Bookshelf, 2007.
- [16] A. Inc. (2016) Dispatch - api reference. [Online]. Available: <https://developer.apple.com/reference/dispatch>
- [17] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.