VRIJE
UNIVERSITEIT
BRUSSEL

Master thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Applied Sciences and Engineering: Computer Science

BUILDING A REINFORCEMENT LEARNING ENVIRONMENT FOR MACHINE INSTRUCTION SCHEDULING

Master Thesis

David Engelman

Academic year 2019-2020

Promotor: Prof. Dr. Ann Nowé
Advisor: Denis Steckelmacher
Science and Bio-Engineering Sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad Master of Science in Applied Sciences and Engineering: Computer Science

BUILDING A REINFORCEMENT LEARNING ENVIRONMENT FOR MACHINE INSTRUCTION SCHEDULING

Master Thesis

David Engelman

Academiejaar 2019-2020

Promotor: Prof. Dr. Ann Nowé
Advisor: Denis Steckelmacher
Wetenschappen en Bio-ingenieurswetenschappen

This master's thesis came about (in part) during the period in which higher education was subjected to a lockdown and protective measures to prevent the spread of the COVID-19 virus. The process of formatting, data collection, the research method and/or other scientific work the thesis involved could therefore not always be carried out in the usual manner. The reader should bear this context in mind when reading this Master's thesis, and also in the event that some conclusions are taken on board.

Acknowledgments

I would like to start by thanking my supervisor, Denis Steckelmacher, for all the advice he gave me during the course of my work. Thank you for all the comments and ideas you brought up during the redaction of this thesis and for always being available for any doubts and questions I had.

I would like to thank the Vrije Universiteit Brussel and in particular the AI Lab that introduced me to the field of Artificial Intelligence that was completely new to me at the beginning of my master. During these 2 years, I had the chance of meeting passionate professors and teaching assistants who allowed me to better understand the challenges AI represents nowadays.

I also want to thank the entire faculty of computer science of the Université Libre de Bruxelles where I spent 3 beautiful years during my Bachelor studies. There I learned the essentials skills and knowledge that I needed to dive deeper into the very broad computer science field.

Thank you to my friends Gregory, Alexandre, Pedro, Florian I met during these last 5 years of studies. All the projects we did together would have been a lot less fun without them.

Thank you to Ruben Lejzerowicz and my father to have proofread this thesis and provided me with valuable feedback.

Special thanks to Benjamin, my twin brother, with whom I had the chance to share the same studies. Everything was easier thanks to the support and advice we gave each other throughout our academic journey.

Abstract

When executing a program, several factors will influence its performance. Of course, among them, we can cite the actual code written by the programmers or the hardware that was used. However, it exists more subtle elements that are related to the compiler used to transform code to actual machine instructions. Compilers autotuning is the field of computer science focusing on finding automated methods to improve compilers and has been studied by researchers since the 1960s.

In this thesis, we focus on one particular problem that is solved by compilers, Machine Instruction Scheduling. The order in which instructions are executed by a computer can influence its run time, and nowadays, instruction scheduling is done using complex heuristics. With the aim of finding automated solutions to instruction scheduling, we developed and propose a new Reinforcement Learning environment. Researchers have already shown, in the past, that Reinforcement Learning agents were capable of outperforming compilers [MMB02]. However, no recent work has tried to automate the instruction scheduling task on modern processors. By proposing a new environment compatible with LLVM, a modern compiler, we hope to revitalize interest in this field.

This master thesis describes how our environment has been implemented and the challenges that we faced during this process. We show it is able to produce a new valid rescheduled program while having influence on its speed. Finally, we highlight how challenging the task of learning in our new environment is, and discuss why current Reinforcement Learning agents seem to not to be able to learn in such a complicated environment. We then propose future research directions that should lead to agents able to learn in our environment, leading to a complete system, able to automatically schedule the instructions in a program.

Introduction

In modern computer systems, microprocessors perform operations by executing a sequence of basic instructions, such as additions, multiplications, memory read and writes, and jumps. Due to performance and historical reasons, several instructions perform the same tasks (variants of an addition, for instance), and some instructions can be reordered without changing the output of the program.

While the precise choice of instructions, and their ordering, might not change the semantics of the program or the results it produces, it influences the speed at which the program runs. Finding the optimal sequence of instructions accomplishing a task is, therefore, crucial to obtain optimal speed or maximum energy efficiency. However, instructions ordering is a monumental challenge. Compiler architects have designed advanced heuristics for over more than 50 years, but the problem is not truly solved yet.

The goal of this master thesis is to design an environment that could be used by Reinforcement Learning (RL) algorithms to tackle this Software Engineering problem.

This Artificial Intelligence technique allows a computer agent to learn how to perform a task in order to maximize a reward. This very flexible framework can be applied to the Machine Instruction Scheduling problem by considering that the agent is the compiler, the task is the production of sequences of instructions, and the reward is built using the performance measure resulting from program runs.

Problem Statement

To be able to solve a problem using Reinforcement Learning the first challenge is to represent the task we want to solve, in our case, instruction scheduling. This is the role of the environment. To our knowledge, there is no such environment available (at least publicly). The second challenge is to learn the environment using a Reinforcement Learning method.

In this thesis, we focused the majority of the efforts in designing a new environment that would allow the use of Reinforcement Learning methods to learn this problem. To this end, the environment built in this thesis is based on LLVM, an Open-Source compiler, that takes C or C++ code as input and produces LLVM intermediate instructions, that

are then transformed into machine instructions executable on most operating systems and microprocessor architectures. However, designing such an environment is a challenging task. Indeed, many available environments nowadays represent games or robotic tasks and can be represented through images or coordinates. In our case, the main challenges to which we propose a solution in this thesis are:

1. The representation of an LLVM program: How do we transform an LLVM intermediate instruction file to something that is understandable by a computer agent and what information needs to be extracted?

2. The actions set available to the agent: Unlike in a "game" setting this is not straightforward. What are the actions that an agent can make when presented with an observation of the LLVM program to reschedule the instructions?

3. The creation of a new program resulting from the agent actions. How do we recreate from the agent's choices a new program that can be compiled and run? This is not as trivial as just rewriting the instructions reordered and different problems had to be solved.

4. Finally, how do we evaluate the produced program? This is a crucial question as without this last step it would not be possible to give a reward to the agent, compromising the learning procedure. Different methods will be presented and assessed including the possibility of evaluating the program on a remote device (different than the one used by the learning algorithm).

The environment will reorder instructions of an original program by recreating a new program from scratch using the instructions of the original one. This is done by asking a question at each step to the agent: "do you want to schedule instruction x ?" with x being one the instruction of the program we want to reschedule. The agent will have 2 possible answers: yes or no. If the agent decides to schedule the instruction, it will be added to the program. Otherwise, another instruction will be proposed to the agent. This will lead to the creation of a new rescheduled program. Of course, we will need to make sure that the behavior of the original program is maintained.

After having implemented our environment we will verify that it has an effect on code performances and as such can be used to learn how to produce more efficient code.

The last challenge we will try to solve in this thesis is the use of our environment by reinforcement learning. The 3 main challenges we will face are:

1. The environment is partially observable. For example, the agent does not observe all the instructions that have already been placed in the program or the schedulable instructions that are yet to be placed. Therefore, the same observation in our environment can lead to different rewards which is challenging for reinforcement learning methods.

2. The sparsity of the reward. Due to its nature (i.e. representing a program speed), it can only be produced at the end of an episode, when the program has been

reconstructed. Because choosing the order of the instructions in a program is done one instruction at a time, and programs typically consist of tens of thousands of instructions, our agent receives an informative reward about 1/10.000th of the time.

3. The time of a learning episode (i.e. one complete rescheduling). The evaluation of a program, which is essential to produce a reward, requires multiple runs of it which will have an impact on the training duration.

With these challenges in mind, 2 different Reinforcement Learning algorithms, DQN and PPO will be tested on our environment.

Structure of this document

This master thesis is divided into 7 different chapters. The first 3 chapters present some background knowledge. Chapter 1 gives some background on compilation in order to set the context in which instruction scheduling takes place and explains the key elements needed to transform code into an executable. The second chapter, about reinforcement learning, presents PPO and DQN methods to highlight their differences before giving some background on Markov Decision Processes and Partially observable Markov decision processes for a better understanding of the challenges we faced. Chapter 3 presents related work that focuses on automatically improving compilers.

Chapter 4 is the core of our contribution, explaining and detailing all the choices that have been made to implement our Reinforcement Learning environment. In chapter 5, we provide evaluations of our environment using different types of performances to demonstrate its behavior and usefulness. Finally, in chapter 6, we present our results obtained by Reinforcement Learning algorithms and discuss the potential improvements that can be made.

Contents

Chapter 1

Compilation

The compilation is the process of transforming a source language text, the code written by a programmer, into a target language text understandable by a computer. Of course, we want to keep the semantics of the original text during this transformation. This so-called compilation process is performed by compilers and many challenges arise during this process.

As this thesis focuses on presenting a new solution to the instruction scheduling problem, this chapter will mainly focus on the parts needed to better understand the motivation and challenges of this problem. Furthermore, we will also discuss the LLVM compiler and its specificity as it is the one chosen to build our environment on.

1.1 The Compilation Process

In this section, inspired from lectures notes [Rog15; Ver09] we will go through the steps of the compilation process that are needed to transform a source code to the target code.

To translate a source code the compiler first needs to analyze it to extract all the language elements. This analysis stage can be divided into 3 steps:

1. **Lexical analysis**: It is performed by the lexical analyzer. The source code, written by a programmer and that serves as input to the compiler, can be viewed as a raw set of characters. During this first compilation step, the job of the lexical analyzer is to transform these characters into *tokens*. The *symbol table* can also be accessed or consulted by the lexical analyzer to store/retrieve information about the source language concept such as variables, function, and types. Finally, some properties can also be attached to a token. For example, a <NUMBER> token will be associated with a value property.

2. **Syntax analysis**: This analysis is performed by the *parser*. Its task is to find the links between the tokens and regroup them into statements (the sentences of a language). It verifies whether or not the sequence of tokens is correct according to the rules of a specific language. The links between the tokens are represented using a tree structure called the *parse tree*.

3. **Semantic analysis**: Using information from the parse tree and the symbol table, the compiler can now associate elements together and perform additional checks. Are variables declared at the right place (i.e. before their usage), are the functions called with the correct number of parameters, etc. At the end of this third step, the compiler has generated an internal representation of the program.

After this analysis, all the errors related to the language would have been detected and reported to the programmer. The compiler is now ready to generate the actual machine code that will be executed.

Similarly to the analysis step, this task can be divided into 3 sub-tasks:

1. **Intermediate code generation**: The compiler transforms the internal representation of a program into an intermediate language. This intermediate language is different depending on the compiler used. More details about the intermediate representation used by the LLVM (the IR representation) will be given in section 1.4.

2. **Optimization**: Using the intermediate representation the compiler tries during this phase different modifications of the code to obtain a similar but faster version of the program. Optimizations are often performed by different passes. Different types of optimizations exist (e.g. loop optimizations, local optimizations, etc.) and some of them require more information than others. Usually, the compilers allow to decide the level at which we want the code to be optimized. **This is during this phase that the compiler may decide to reorder instructions**.

3. **Code generation**: This is the final job of the compiler. The intermediate code representation, possibly optimized, is translated to assembly instructions for the specific microprocessor the program is compiled for. One of the challenges here is register allocation: deciding which registers to use when given that a piece of code may use many more local variables than the number of registers available on the target microprocessor.

1.2 Instruction Scheduling

As mentioned in the previous section, the compiler may decide to reorder some instructions to achieve better performance and faster run time. This process is called instruction scheduling. It can be viewed as finding the optimal order of intermediate code instructions to maximize parallelism and minimize the number of machine instructions generated. In this section, we will dive deeper into this topic and provide the elements needed to understand how it is done and, more importantly in the case of this thesis, what are the essentials elements needed to build an RL environment able to reorder instructions.

1.2.1 Basic Blocks

The intermediate code generated by the compiler is composed of a set of functions, itself composed of sequences of *basic blocks*. To define more formally what a basic block is we

first need to look at the definition of a leader instruction [Ver09]:

Definition 1. *Let s be a sequence of intermediate code instructions. A **leader instruction** is any instruction $s[i]$ such that:*

- *$i = 0$, i.e. $s[i]$ is the first instruction; or*
- *$s[i]$ is the target of a (conditional or unconditional) jump instruction; or*
- *$s[i-1]$ is a (conditional or unconditional) jump instruction*

*A **basic block** is a maximal sub-sequence $s[i], ..., s[i+k]$ such that $s[i]$ is a leader and none of the instructions $s[i+j], i < j \leq k$ is a leader.*

1.2.2 Data Dependencies

The big challenge when doing instruction scheduling is to persevere the behavior of the program. To this end we need to respect 2 rules:

1. The set of operations performed on any computation path need to be preserved.
2. Interfering operations need to be performed in the same order.

To understand what interfering operations are, we need to understand the different types of *data dependencies*

1. **True dependence** (read-after-write): A read of variable x must continue to follow the previous write of x.
2. **Anti-dependence** (write-after-read): A write of variable x must continue to follow the previous read of x.
3. **Output dependence** (write-after-write) Writes of variable x must stay in order

Interfering operations contain dependent data.

Note that true dependence is the only kind that cannot be eliminated. Indeed it is possible to eliminate anti and output dependencies by introducing additional intermediate variables.

A common way of representing data dependencies is the use of a directed acyclic graph (DAG). The nodes of this kind of graph are the instructions and the dependencies between the instructions are represented through the edges. The dependencies graph of a basic block needs to be consistent with the 2 rules presented above. An example of data dependencies graph is shown in Figure 1.1. Looking at this graph, instructions 3 and 4 could be swapped around without changing the meaning of the program. On the other hand, instructions 1 and 2 could not be swapped as instruction 2 depends on the write done by instruction 1.

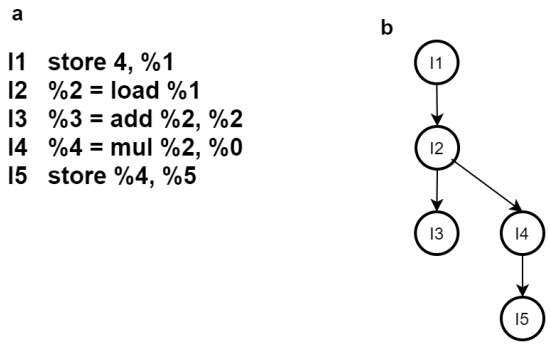


Figure 1.1: Data dependencies graph (b) of instruction block (a).

1.2.3 Pipelining

To understand why the order of instructions matters, we need to introduce pipelining. Modern computer architectures do not execute one instruction at a time. Indeed one single instruction can be decomposed into different stages. As these stages use different parts of the processor, this decomposition allows parallelizing instructions execution (Figure 1.2a) and is called pipelining. One classic stage decomposition is composed of 5 stages: fetch, decode, execute, memory access, writeback to register.

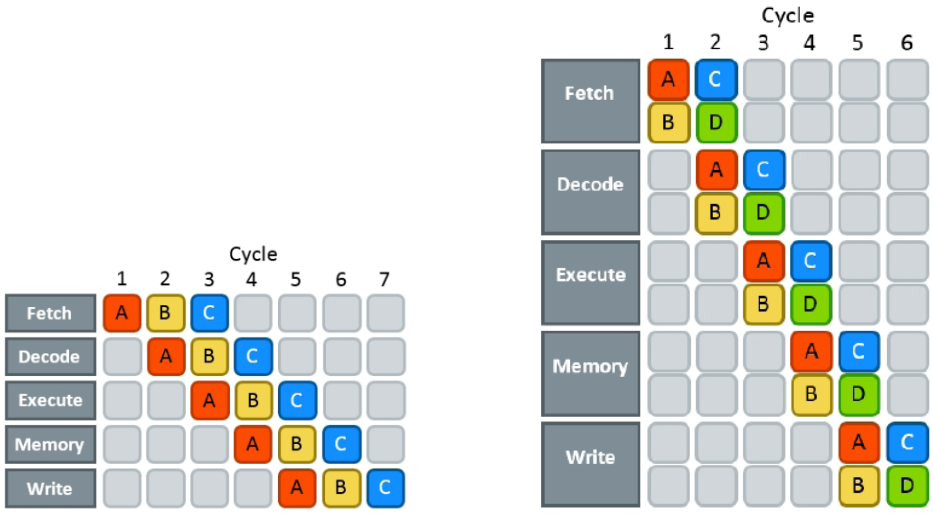
However, this parallelism introduces what is called pipelining hazard. This occurs when some data is needed by multiple instructions (cf. data dependencies) or if a hardware resource is required by different instructions in the same cycle (this is called structural dependencies). To solve this, the processor has no choice but to wait for the hazard to be resolved causing the next instructions to be delayed. By reordering instructions, it is possible to limit these delays, also called "stalls", improving a program speed. To increase parallelism even more, processors use superscalar architectures (Figure 1.2b), enabling them to execute multiple instructions through the same stage.

1.3 Compilers

Nowadays, many different compilers exist. One of the key components of the RL environment that we will design in this thesis is the compiler. Choosing the most appropriate one is then an important task. In this section, we will describe 2 compilers that have been used in the literature to automatically tune compilers and motivate the choice that we made for our contribution.

1.3.1 GCC

The GNU Compiler Collection, GCC in short, is an open-source project developing a family of compilers that can be used on 8 different languages. The languages are C,



(a) (b)

Figure 1.2: Pipelining (a) and superscalar (b) instruction processing. Multiple instructions can be processed during 1 cycle.
`https://techdecoded.intel.io/resources/understanding-the-instruction-pipeline/`

C++, Objective-C, Objective-C++, Java, Fortran, Ada, and Go. GCC also supports different processor architecture. GCC has a relatively monolithic structure, and has been designed to be used as a command-line tool that takes a source file as input, and produces an executable as output. It is possible but difficult to interact with the inner components of GCC or to generate code with GCC from a Python program. Tuning using GCC requires to override the pass manager component as, otherwise, GCC would override the predefined ordering.

1.3.2 LLVM

The LLVM Project is a collection of modular and reusable compilers and toolchain technologies that has been vastly used in compiler autotuning research. It provides an intermediate language, called LLVM-IR, which can easily be parsed, analyzed, and transformed. To this end, some tools (e.g. the llvmlite python library) have been developed and could be useful. Clang, its c/c++ front-end, offers many flags and options to control the target code that is being produced. LLVM, like GCC, is able to target many different microprocessor architectures.

The existence of the LLVM-IR language, the clang interface, and the availability of python tools motivated our choice to use LLVM. Its rich and well-structured documentation was also a determining factor.

1.4 LLVM-IR Language

As mentioned in the previous section, LLVM uses a normalized intermediate representation language. As it will be the input of our RL agent, it is important to describe its form and specificities. LLVM-IR is available through 3 possible forms:

1. In-memory compiler IR
2. Bitcode (can be used to be loaded to a JIT)
3. Human readable format (see below)

Listing 1.1: Hello world module in LLVM-IR in human-readable format. The code is composed of one constant declaration and one function declaration containing one basic block.

```
1  target triple = "x86_64-unknown-linux-gnu"
2
3  @s = private constant [13 x i8] c"hello world\0A\00"
4
5  ; main prints "hello world" to standard output.
6  define i32 @main(i32 %argc, i8** %argv) {
7          %1 = getelementptr [13 x i8]* @s, i32 0, i32 0
8          call i32 (i8*, ...)* @printf(i8* %1)
9          ret i32 0
10 }
11
12 declare i32 @printf(i8*, ...)
```

Looking at its structure, an LLVM code is composed of modules. Each module is composed of functions, constants, and symbol table entries. A function is defined using the "define" keyword and contains the name of the function, its argument list, return type, and can contain a long list of some optional additional information.

The code of each function is contained in basic blocks. These basic blocks are simply separated by using an empty line between them. In each block, we can find the LLVM instructions.

LLVM instructions can be categorized into 4 main categories:

1. **Binary instructions**: They require 2 operands and are responsible for doing most of the computation in a program.
2. **Memory instructions**: These are the instruction that will interact with the memory in order to allocate, read, write, etc.
3. **Bitwise binary instructions**: Bitwise binary operations are used to perform bit-twiddling operations. this type of instruction can sometimes be used to replace others to gain performance. For example, divisions and multiplications can sometimes be performed using shifting operations.

4. **Terminator instructions**: They are always at the end of a block and each block must always end by a terminator instruction. They are responsible for the control flow of the program.

Any LLVM instruction can always be described by its opcode and its operands (possibly none).

Finally, LLVM-IR defined 3 types of identifiers. They can be either global or local:

1. Named identifiers: starting with a "%" or an "@" (e.g. %foo, @DivisionByZero, %a.really.long.identifier)
2. Unnamed identifiers using numeric values (e.g %12, @2)
3. Constants

In the next chapter, we introduce and review Reinforcement Learning, that we will use to automatically tune programs written in the LLVM intermediate representation.

Chapter 2

Reinforcement Learning

Reinforcement Learning (RL) is a field of Artificial Intelligence where an agent learns to perform a task by performing actions in a given environment, while receiving rewards and punishments. The typical RL learning loop is summarized in Figure 2.1. At each step, the agent chooses and performs an action among a set defined by the environment. The environment then decides on a reward for the agent. This reward is the learning signal that will allow the agent to improve. The reward depends on the action chosen by the agent but is also related to the state where the agent was and where he ended when choosing this action. For example, in a maze, deciding to go right when it leads to a treasure is way more rewarding than going left and falling in a pit.

In addition to choosing the reward, the environment is responsible for determining its new states, translated to an observation for the agent, after an action has been performed. The states can be of 2 types: discrete (e.g. the x and y coordinate in a maze) or continuous (e.g. the angle of the wheels, the velocity of a car, etc). Of course, the more states there are, the more challenging the learning task is.

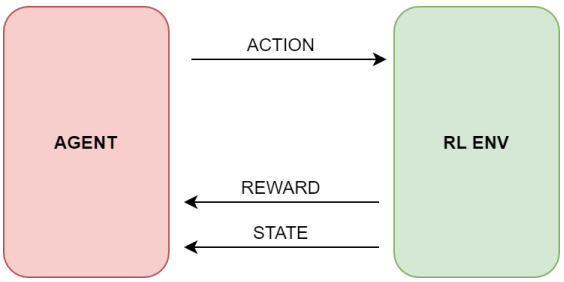


Figure 2.1: RL learning loop. The agent performs an action in the environment and receives a reward and the new state.

Another complication that may arise in an RL setting is the sparsity of the reward signal. Indeed, in some problems, it is not always possible to give a meaningful reward at each step and we may have to delay the reward, sometimes until the end of an episode.

Some techniques have been proposed [And+17; Arj+19] to tackle these sparse reward problem but they cannot be applied to all the tasks.

The end-goal of the agent is to discover which action to take in which state in order to maximize the discounted cumulative reward which is defined as:

$$\sum_{t=0}^{\infty} \gamma^t r_t$$

with γ, called the discounted factor, between 0 and 1, it weights the importance of a reward based on when it was obtained. By discounting the rewards, the action will prefer actions that are rewarded more rapidly.

2.1 Finite Markov Decision Process

To define more formally the concepts presented above, we need to introduce Finite Markov Decision Process (MDP). This framework provides all the necessary elements to be able to learn from feedback. To be defined as a finite MDP, a problem must be able to be broken down into :

1. Discretes time steps $t = 0, 1, 2, 3, ..., n$. At each time step, the agent performs an action and gets a reward. These time steps do not need to specifically define a time. They can, more generally, be viewed as decision moments.

2. A finite set of states $\mathcal{S}$. They are used to send information about the environment to the agent. The agent is able to see the state as an observation.

3. A finite set of actions $\mathcal{A}$ which can be performed by the agent.

4. A reward function $R(s, a, s')$ emitting the reward for the action $a \in \mathcal{A}$ from state s to s'.

5. A transition function $T(s, a)$ returning a probability distribution of the next states that can be reached after performing action A in state S. The probability of reaching s' resulting in a reward r from state s after performing action a at time t:

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

An agent is able to observe the state it is in and the available set of actions. It has no information related to the reward or transition functions. When designing an RL environment we have to carefully design all the above-cited components.

2.2 Partially Observable Markov Decision Process

We have seen in the previous section how a problem can be expressed as an MDP. However, it is not always possible for an agent to be sure in which state it is standing. In this case, the Markovian Decision process becomes a Partially Observable Markov Decision Process (POMDP). As an example, we can think of a shooter game where a player cannot see behind him. Because of this partially observable world, the same observation may correspond in reality to 2 different states that would lead to different rewards. Indeed, let's look at 2 situations. (1) The agent is standing behind an enemy that is not moving, (2) The agent is standing behind an enemy that is not moving but there is also an enemy behind the agent. In these 2 situations, the agent sees the same observation but the real states are not the same.

More formally, a POMDP assumes the existence of a state s in the environment, but that is hidden from the agent. Instead, the agent observes an observation $o = O(s)$, produced by the observation function. The reward given to the agent still depends on the unobservable state s, $r = R(s, a, s')$. This means that the agent does not observe everything that was used to compute the reward it receives, which potentially prevents it from fully explaining its reward (after identical observations, the reward is sometimes high sometimes low, depending on factors the agent does not observe). The environment we designed in this thesis (see chapter 4) is partially observable, as at each learning step the agent is not provided with the complete program it is rescheduling (which would have been very complex to represent), but only a restricted view of it.

Different methods have been proposed in the literature to help to solve POMDP environments. In the early days of this research topic, researchers tend to propose algorithms for finding the exact optimal solutions [KLC98; Cas98]. However, these solutions are not suited when the number of possible states becomes very large. That is why researchers started to develop approximation methods.

As it is not possible for an agent to observe the complete state it is in, it would be very useful for an agent to remember the previous observation he made to better understand the states. That is the reason why methods that make use of recurrent neural networks (RNNs) to learn a policy have been proposed [Bak02; HS15; ZS15; Zhu+17]

These types of networks are designed to incorporate previous observations into their current decision. This is done by processing the information in a sequential way. When feeding an input x into an RNN the output y depends not only on x but on the whole input history. This is done by keeping a hidden state vector h in the network that is updated when a new input is fed into the network. The hidden state is updated using:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{2.2}$$

The first term of the addition gathers knowledge about the past and the second introduces the newly fed sample. The result of the addition is then squashed by using tanh as

the activation function. Nowadays, LSTMs tends to be preferred over RNNs. An LSTM is a particular kind of RNN with a modified update rule for the hidden state which tends to work better in practice.

In the next section, we will present 2 different kinds of RL methods that have been developed to learn a policy in an environment.

2.3 Deep Q-Learning

One of the learning algorithms that will be tried on our environment is DQN. Before diving into this method to understand how and why it works we need to introduce the method it is derived from: Q-learning.

2.3.1 Q-learning

Q-learning was introduced by Watkins and Dayan [WD92]. It can be classified as a value-based method. Indeed, the goal here is not to directly learn the optimal policy but rather to associate values to each (state, action) pairs. When this is done, these values are used to derive the optimal policy, they are called Q-values.

At the beginning of the learning process, all the Q-values are set to an initial value, often zero. The goal of the learning process is to update them so that they can be used to derive the optimal policy. After each transition we can update the Q-value of a state action pair using the q-learning update rule defined as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.3}$$

The update is using the TD-error. α is the learning rate, controlling by how much the Q-values are updated and γ is the discounted factor weighting the importance of early rewards. After training, using the updated Q-values we can easily derive the optimal policy function defined as:

$$\pi^*(s) = \operatorname*{argmax}_a Q(s, a) \tag{2.4}$$

Q-learning works to solve small environments but has one obvious limitation: it has to store all the Q-values in memory which means $number\_of\_states \times number\_of\_actions$ values. This number can rapidly grow and become way too big to be manageable. A solution to this problem is the use of function approximator.

2.3.2 DQN: The Introduction of Neural Networks

DQN introduced by Mnih et al. [Mni+13] uses a neural network as function approximator to tackle this problem. Using this method it is no longer necessary to store all the Q-values. Indeed, using neural networks, it is now possible to approximate them. We can

update the weights of this newly introduced network by applying gradient descent to minimize the TD-error:

$$\alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a, \theta_t) - Q(s_t, a_t, \theta_t)) \tag{2.5}$$

where θ_t are the network parameter at time t and

$$r_{t+1} + \gamma \max_a Q(s_{t+1}, a, \theta_t) \tag{2.6}$$

is called the target.

After the introduction of DQN, different improvements have been proposed in the literature.

2.3.3 The Target Network

One problem that arises from vanilla DQN is that it uses the same network parameters θ to predict the target and the estimated Q-values. In other words, we are chasing a target that is moving away from us at each parameters update. Also, updating the parameters for one action will have an influence on others. To solve this, Mnih et al. [Mni+15] proposed the introduction of a second network called the target network. The method now has one network to estimate the Q-values, the q-network, and the newly introduced target network to estimate the target. The weights of this target network are not updated at each step. Instead, the weights of the q-network are periodically copied to the target network to solve the moving target issue.

2.3.4 Double DQN

Another observation that has been made is that using neural networks may cause overestimation (sometimes call upward bias) of the Q-values This comes from the fact that we are assuming that the best action for the next state is the action with the highest Q-value. This is not always the case as the predicted Q-values can be noisy (they are estimations), especially at the beginning of the training.

Double DQN is a method that was introduced by Van Hasselt et al. [VGS16] to solve this overestimation problem. The solution proposed is to separate the action selection process from the evaluation process. The target computation now uses both the target network and the q-network to become:

$$r_{t+1} + \gamma Q(s_{t+1}, \operatorname*{argmax}_a Q(s_{t+1}, a, \theta_t), \theta_t^-) \tag{2.7}$$

where θ^- are the target network weights and θ are the q-network weights.

2.3.5 Dueling DQN (DDQN)

Wang et al. [Wan+15] proposed a new network architecture called the dueling network architecture. The idea behind it is that a Q-value, representing how good it is to perform an action in a given state could be decomposed into 2 different part:

1. A Value function $V(s)$ estimating the quality of a state independently of an action.
2. An action advantage function $A(s, a)$ that can evaluate each action in a given state.

Using this decomposition we can define a Q-value as:

$$Q(s, a) = V(s) + A(s, a) \tag{2.8}$$

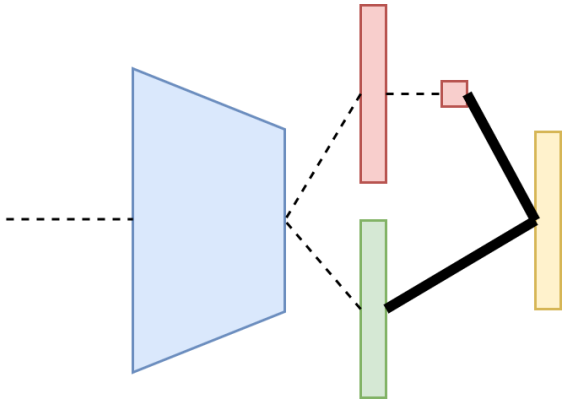


Figure 2.2: DDQN architecture composed of one common stream (in blue), a state value function estimation stream outputting a scalar (in red), and the action advantage estimator (in green). The two streams are combined at the end to obtain Q-values for each action.

By decoupling the 2 functions DDQN it is no longer necessary to calculate the effect of taking each action in a given state to know how good it is. This is useful when some outcome in a state is independent of the action taken. (e.g. all the actions result in the agent failure/success).

The Q-value of a state action pair can be found by combining the the value and action advantage function in this way:

$$Q(s, a, \theta, \alpha, \beta) = V(s, \alpha, \beta) + (A(s, a, \theta, \alpha) - \frac{1}{\mathcal{A}} \sum_{a'} A(s, a', \theta, \alpha)) \tag{2.9}$$

where θ represents the weights of the common stream part, α represents the weights of the action advantage stream, and β represents the weights of the state value estimation stream. $\mathcal{A}$ is the set of available actions. We might have been tempted to simply define $Q(s, a, \theta, \alpha, \beta) = V(s, \alpha, \beta) + A(s, a, \theta, \alpha)$ but this would have made the backpropagation impossible as we would not have been able to find $V(s)$ and $A(a, s)$ given $Q(s, a)$.

2.3.6 Experience Replay

The vanilla way of training an agent is to perform an action and then, directly based on this experience, an (s, a, r, s') tuple, update the RL algorithm. By doing this we can highlight 2 potential problems:

1. The agent tends to forget past experiences as we are constantly overwriting them with the latest one. Experiences can only be used once.

2. There is a high correlation between the sequence of experiences.

To solve these 2 problems the experience replay mechanism [Lin93] can be used. At first, this method uses a replay buffer that can be seen as the memory storage. Instead of using each experience directly, they are stored in this replay buffer so that the agent does not forget them. The RL algorithm is not updated at each step anymore. Indeed, after a fixed number of steps (this is decided by the programmer) a batch of experiences sampled at random from the replay buffer is used to update the algorithm. By doing this we break the correlation between the experiences.

It is possible to further improve this experience replay mechanism. Indeed some experiences might be more valuable than others. For this reason, it could be useful to weight experiences and give a higher chance to more informative experiences to be drawn when replaying. This is exactly why prioritized experience replay (PER) was introduced [Sch+15a].

The priority of an experience is defined as:

$$p_i = \mid \delta_i \mid + \epsilon \tag{2.10}$$

This definition uses the TD-error δ to quantify how much knowledge an experience can bring and, a constant epsilon is added so that all experiences have a chance of being chosen. However, we cannot just decide to replay the experiences with the highest priority every time. Indeed, the choice of an experience needs to keep incorporating some stochasticity. For that reason, the probability of choosing an experienced to be replayed was defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.11}$$

a is used to control how the prioritization is used. With $\alpha = 0$, we are in the uniform case and with $\alpha = 1$, we always select the experience with the highest priority.

One complication of using PER is that our experiences are not sampled from a uniform distribution anymore. PER introduced a bias toward highly prioritized experiences and this could lead to over-fitting as the weights of our model will be mainly updated using these experiences.

To avoid this, the weights update is done using importance sampling (IS) weights:

$$w_i = (\frac{1}{N} \times \frac{1}{P(i)})^{\beta} \tag{2.12}$$

Using IS weights, the model parameters associated with low priority experiences will be more updated than the ones associated with the highest priorities. The β parameter control by how much we want this IS correction to be used.

Experience replay can be used with not only DQN and its extensions describe in this section but with any offline RL algorithm.

2.4 Proximal Policy Optimization

We just saw an example of a value-based method. On the other hand, it exists another kind of RL algorithms that directly learn the optimal policy. These are categorized as policy-based methods. Proximal Policy Optimization (PPO) is one of them. As these methods are learning in an online setting they cannot make use of experience replay. That is why they are usually less sample efficient than offline methods (e.g DQN).

PPO is part of the policy gradients methods family. The goal of these methods is to train a neural network to directly model the best action probabilities. It is an iterative process where at each step we use the experiences to update the weights of the network until it has converged to the optimal policy π^*.

2.4.1 Vanilla Policy Gradient

Vanilla Policy Gradient (VPG) [Sut+00] tries to maximize the sum of the cumulative discounted rewards. More formally the objective function (i.e loss) of our model is:

$$L^{PG}(\theta) = L^{PG}(\theta) = E_t[log\pi_\theta(a_t \mid s_t) \times A_t] \tag{2.13}$$

The log term represents the probability of taking a_t which is obtained from the policy by taking the observed state as an input. The second θ A_t is an estimate of the advantage function (similar to what we saw in section 2.3.5). To compute this advantage it uses the difference between the discounted cumulative rewards and the estimate of the cumulative reward from the current state. The advantage function is defined as:

$$\sum_{t=0}^{\infty} \gamma^t r\_V(s) \tag{2.14}$$

So, it tells us if the action the agent took was better or not than expected. If the output of this advantage function is positive the probability of taking the action a_t will increases (as the gradient will be positive) which is exactly what we want.

Difficulties come from the step size taken during the gradient ascent process. Indeed, if it too big, we will observe too much variability as our estimate of A_t could become completely wrong. Inversely, if the steps are too small, it will take too much time to converge to the optimal policy.

2.4.2 TRPO

To try solving this stability issues the Trust Region Policy Optimization (TRPO) method was introduced by Schulman et al. [Sch+15b]. In their method, they modified the objective function by replacing the log term by a probability ratio r_t of the current policy over the old one. More formally:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta old}(a_t \mid s_t)} \tag{2.15}$$

This probability ratio is greater than 1 if the action is more probable considering the current policy and between 0 and 1 if it was more probable using the old policy. By introducing another policy in the loss term, it allows to better constraint the changes . Using this new ratio term we can modify the VPG objective function (eq. 2.13) to:

$$L^{CPI}(\theta) = E_t[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta old}(a_t \mid s_t)} \times A_t] = E_t[r_t \times A_t)] \tag{2.16}$$

However, using this new objective function could still lead to large step size. Indeed, if the upper term of the ratio is much bigger than the lower term we may sill have variability problems. The solution proposed in TRPO is to limit the step size without changing the objective function but rather uses KL-divergence constraints to make sure our current policy stays in a region close to the old one. However, this KL-divergence constraint provide additional complexity to the optimization process and can lead to undesirable training behavior.

2.4.3 PPO

PPO directly incorporates the policy variation constraint in the optimization function [Sch+17]. To this end, it makes use of a new objective function called the *clipped surrogate objective function*. This function will improve learning stability by limiting the change made to the policy by clipping the values.

The new objective function used by PPO is:

$$L_t^{CLIP}(\theta) = E_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{2.17}$$

To understand how this function constraints the policy updates, it is useful to decompose its uses in 2 cases (see figure 2.3) :

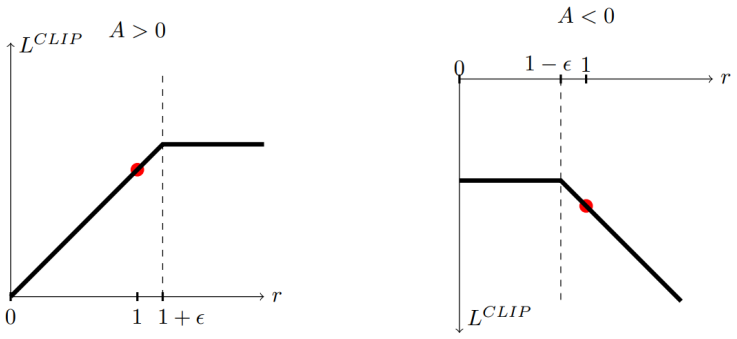


Figure 2.3: Effect of the Clipped surrogate function depending on the value of A_t. Plot taken from the original paper.

1. $A_t > 0$: The action picked was a good choice. The loss function is clipped when r is getting too high (the action just became more probable) to avoid big shifts of the policy.

2. $A_t < 0$: The action picked was a bad choice. Here the values are clipped when r is getting small, meaning that the action became less probable.

Finally to fully train an agent we need to incorporate the loss of the value function network used in the computation of the A_t (see eq. 2.14) and a third term that will encourage exploration using entropy. The complete loss is:

$$L_t^{PPO}(\theta) = E_t[L_t^{CLIP}(\theta) - c1L_t^{VF} + c2S[\pi_\theta](s_t)] \tag{2.18}$$

The final PPO network architecture is composed of 2 heads (i.e. parts). The V(s) part to approximate the expected cumulative reward from a state s and the policy selecting the actions sampled from a Gaussian distribution.

This section concludes the Reinforcement Learning chapter. In the next and last chapter we will see what type of research have already been done to try to automatically improve compilers.

Chapter 3

Related Work

Using Artificial Intelligence in order to improve or tune compilers is a big challenge that could have a big impact on multiple fields of computer science. This task, more formally called *compilers autotuning*, is defined in the literature review done by Ashouri et al. [Ash+18] as:

Definition 2. *a methodology where there is some model (can be a search heuristic, an algorithm, etc.) that infers one or more objectives with minimal or no interaction from a user.*

This Chapter is built using this literature review as the central source to understand the state of the field.

To find the first use of automatic methods used as an attempt to improve compilers and solve their complex optimization problems we need to go back to the 1990s. One of the first works on this subject by Whitfield et al. [WS91]. The authors proposed a new specification language, General Optimization Specification Language (GOSPEL), and a tool used to analyze it called Genesis to improve the code produced by a compiler. They demonstrated that using Genesis, it was possible to select the optimal least expensive method to reduce the cost of the optimizations.

3.1 Source Code Feature Extraction

As we have seen in the compilation Chapter 1, the compilation and optimization process is a complex task. If we want a computer to be able to automatically optimize some code, the first step is to be able to transform this code into a representation that can be understood by a machine learning algorithm. Different approaches can be used and combined to achieve this.

3.1.1 Static Analysis

These techniques can be used to collect features defining a source code without the need of running it and independently of the architecture that is used to run it. For example, the

parsing of IR representation can be performed to extract the functions, basic blocks, and instructions. It can also be useful to parse the front-end code (i.e. the source language) or the back-end code (i.e. the target code). Two types of features can be extracted:

1. Source code features (SRC): These features can be considered as some raw features extracted from the source code. For example, the name of the function the compiler is currently optimizing or the number of instructions of a given basic block. Many SRC extractors have been proposed in the literature. Among them, we can cite Fursin and al. [Fur+11] who proposed a GCC plugin to extract source code features .

2. Graph-based Features: Two different graph-based structures can be used to represent and characterize a code at different levels and brings different types of information. Indeed we defined in the compiler chapter: (1) dependency graph, (2) control flow graph (CFG). These graphs can be used to autotune compilers. In 1997 already, Koseki et al. [KKF97] used CFG to measure the efficiency of loop unrolling. At the time, the idea was to improve parallelism. In the past decade, Park et al. [PCA12] proposed a machine learning method using graph-based features to predict good optimization sequences.

3.1.2 Dynamic Characterization

In some cases, it may be useful to run a program to obtain some additional features. This, as opposed to static characterization, is called dynamic characterization. The goal here is to collect what is called Performance Counter (PC). These PCs can be of great help when we want to look for application bottlenecks. Some information related to how an application uses caches, which can be important to obtain efficient optimizations, can also be captured by these PCs. The latest can be classified into 2 mains classes:

1. Architecture Dependent Characterization PCs: Some useful characterizations may be useful to extract but are architecture-dependent. This means that their value cannot be compared across 2 different architectures. This is the case for memory footprints or information related to caches.

2. Architecture Independent PCs: Often referred to as feature vectors, these PCs summarize the performance of a program at run time (e.g. caches hits). As mentioned, caches and memory characterizations are often architecture-dependent as they can vary depending on which architecture the program runs. To tackle this problem, instrumentation tools have been developed to extract these run time architecture-dependent features in a way to transform then into characterizations that do not depend on the architecture anymore as long as the architectures use the same instruction set architecture (ISA).

Finally, it is interesting to note that Dynamic Characterization can be used in combinations of Static Analysis to obtain more features. This is referred to as Hybrid Characterization.

3.2 Machine Learning Methods

Now that we have the different approaches in the literature to gather features about some code to be optimized, we will go through some of the machine learning (ML) methods that researches have applied to improve compilers in an automated way.

3.2.1 Supervised Methods

Many supervised learning methods have been applied to improving the efficiency of compilers. Decision Trees, which is one of the most used supervised methods, have been used by Fraser et al. [Fra99] to produce a compressed version of the IR code. Using decision trees, they were able to improve in an automated way instructions encoding by splitting it into different streams. More related to what we are trying to achieve in this work, Moss et al. [Mos+98] tried 4 different supervised methods to tackle the instruction scheduling problem, more specifically, straight-line code scheduling. Cavazos and Moss [CM04] also worked on block scheduling by applying inductions techniques to come up with heuristic allowing to determine which block should be scheduled. This work was built upon the JIT Java compiler.

3.2.2 Unsupervised and Objective Maximization methods

Unsupervised methods are often used in the ML world to identify patterns or clusters in the data. These types of methods can be applied to a lot of domains(e.g. fraud detection, social networks, etc). Unsupervised methods can also be used for compilers autotuning. Martins et al. proposed a clustering method to find the best combinations of optimizations to be applied to each function. Evolutionary algorithms are another kind of unsupervised method that researchers have explored. As we just saw that optimization selection can be learned using clustering methods, Kulkarni et al. used evolutionary methods to tackle the problem. In one of their work [Kul+13], they used neuro-evolution [SM02] to decide whether or not a method should be inlined. They were able to come up with new heuristics that outperformed manually-constructed ones leading to up to 114 % accelerations. In another publication [KC12], they proposed an effective way of learning the phase-ordering optimization problem by formalizing it using Markov Process and solving it using neuro-evolution as in their last cited work.

3.2.3 Reinforcement Learning Methods

The last machine learning methods that we will cover in the section is RL. This is the method we chose to use for our contribution. As shown in this section, researchers have already tried to use RL methods to improve compilers. However, in comparison to the supervised and unsupervised methods, we can note that RL techniques have been much less explored in the literature.

In 2002, McGovern et al. [MMB02] were the first to present an environment as well as a training strategy by optimizing the state value function using the temporal difference

algorithm [Sut88]. In their method, the agent learns the difference of the returning from choosing an instruction A over instruction B. The agent made its choices using a feature vector using the partially scheduled program. At the time, their features were directly derived from the compiler used. The reward signal was built by comparing the number of cycles to execute a block using RL versus the number of cycles without it. The reward was normalized by the number of instructions in the given block. They demonstrated that the RL agent was capable of outperforming a scheduler used in comparison in multiple C applications. In the same paper, the authors proposed a hybrid method that used RL in combination with rollouts leading to even better results.

Six years later, in 2008, Coons et al. [Coo+08] used an RL method called NEAT, to present a method to generate specialized heuristics for instruction placement on EDGE processors. They showed that using NEAT, their method was able to automatically tune and outperform hand-made heuristics for instruction placement.

To end this section, We wanted to mention a very recent work published on the Google AI blog post[1] on chip design using RL. Designing the layout of chips blocks (referred as floorplanning) is a big challenge. Using deep reinforcement learning (DRL), they were able to train an agent that learns how to place chip blocks and can generalize . Even if this work is not related to compilers autotuning, it illustrates that AI can be used to tackle hardware and software design problems. As the need for always better software and hardware will only increase with time this is great news for the future.

[1] https://ai.googleblog.com/2020/04/chip-design-with-deep-reinforcement.html

Chapter 4

Environment Implementation

In this section, the core element of this thesis will be presented. We will describe all the choices made to implement our RL environment. The objective was to implement an environment that takes LLVM IR representation code as input and can rebuild a new rescheduled version of the LLVM program. For every function in the program, we iterate over its basic blocks of instructions. For each basic block, we sequentially present its instructions to the agent that can decide whether to emit the instruction in the output program, or ignore it and keep it for later. This allows the agent to progressively choose in which order to emit instructions of the program.

The agent must be able to decide whether or not it wants to schedule the showed instruction now or not. If the agent chooses to schedule, the environment will update the new program. This process is repeated until all the instructions of the original program have been rescheduled. It is summarized by Algorithm 1. More details on how instructions are shown to the agent and how the new program is built will be given later in this chapter.

Algorithm 1: Environment High Level Algorithm

```
Initialize 𝓘 to the set of instructions from the Input;
Initialize 𝓟 to an Empty set;
while 𝓘 not empty do
    shows one instruction i from 𝓘;
    if agent choose to schedule i then
        𝓟 += i;
        𝓘 -= i;
Recreate valid executable program from 𝓟;
Execute and time the new program to get a reward 𝓟;
```

To make sure that the new program built by the agent is correct and is not affecting the behavior of the original one, the environment will only propose schedulable instructions. It can present multiple times the same instruction until the agent decides

to schedule it, limited to 4 trials (after this the instruction is directly placed in the program in construction). After the agent has decided on a new order of instructions, the environment will be responsible for reconstructing the new program. This process is not trivial and will be detailed in a dedicated section. Finally, the program is compiled by the environment using clang to be timed during its execution. This time is used to give a reward to the agent.

From this description, we can identify 5 main components that are needed in our environment (The model architecture is presented in Figure 4.1):

1. **LLVM Parser**: To parse the input LLVM-IR program to a set of instructions.
2. **LLVM Dependency Graph Builder**: Responsible of building the dependency graphs of each basic block.
3. **LLVM Scheduler**: Responsible for handling the LLVM code to decide which instructions can be shown to the agent without breaking the original program using the LLVM Dependency Graph Builder.
4. **LLVM IR Builder**: To create a new valid program from the choices made by the agent.
5. **LLVM Timer**: To execute and time the new program.
6. **Gym Interface**: To present actions, and observation to the agent, using the Gym interface (a library of RL environments developed by openAI), to the agent.

In the next sections, we will present how all these modules were built, highlighting the difficulties and choices that had to be made.

4.1 The LLVM Parser

The first task that needs to be performed by our environment is the extraction of all the instructions from the input code so that the agent can reschedule them. This is the role of the LLVM parser. When parsing the LLVM-IR, we need to maintain a certain structure. First, the order of basic blocks in the functions has to be preserved and we need to know when a new function begins. This will be useful when recreating the output program. For this reason, the output of the parser is a 3-dimensional array containing all the functions (first dimension), basic block inside them (second dimension), and the instructions (third dimension).

In a first attempt at parsing the LLVM code, we used the llvmlite Python library. It provides tools to directly iterate over functions, blocks, and instructions, implementing all the parsing logic for us. Unfortunately, we discovered that some metadata elements were renamed using this library. This caused problems for the recreation of the program and decided it was easier to rewrite our own parser. The pseudo-code of our parser is

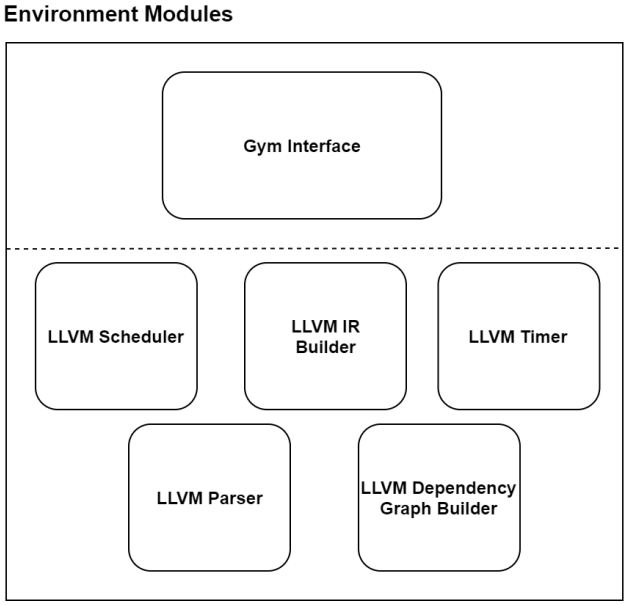


Figure 4.1: Modules composing the Environment

shown in Algorithm 2.

The detection of a function start is easily done as all function declarations start with "declare" keyword (see section 1.4). Looking at blocks, we can use the empty line separating them to detect their start/end. Finally when dealing with an instruction line, the parser needs to parse it to retrieve key elements that will be used by the environment. Four elements are retrieved:

1. The raw instruction line.

2. The opcode of the instruction. This will be useful to extract the type of instruction (e.g memory, binary instruction, etc).

3. The operands of an instruction. This will be used to determine if an instruction can be scheduled without breaking the original code behavior.

4. The name of the variable storing the result of the instruction (if any). This will be useful when we recreate the newly rescheduled program.

Having extracted all this information, we now look at how it will be used to decide which instructions can be shown to the agent.

Algorithm 2: LLVM Parsing Algorithm

```
input  : the LLVM raw code
output: the set of all the code instructions F
initialize empty functions container F;
for line in code do
    if function start then
        initialize new Block container B;
        initialize new instructions container I;
    else if end function then
        add I to B;
        add B to F;
    else if block start then
        add I to B;
        initialize new instructions container I;
    else
        parse instruction line;
        add instruction to I;
```

4.2 Instructions Dependencies

Our environment will produce a new program by changing the order of the instructions. However, we need to be sure that it does not break the original behavior. For this reason, the first operation performed after the parsing is the creation of the dependency graph. The algorithm used to build it is shown in Algorithm 3. Two types of dependencies relations are created:

1. **Instruction dependencies**: This concerns the handling of all the instructions independently of their type. We can say that instruction 1 depends on instruction 2 if instruction 1 is using the result of instruction 2 in its operands.

2. **Memory related dependencies**: This concerns the handling of a defined subset of memory instructions that need to be treated differently from other instructions. For all memory instructions, except the "store", we need to add additional dependencies to previous memory instructions. Table 4.1 lists the memory instruction that are considered by the environment and the memory dependencies are summarized in Figure 4.2. The goal here was not to come up with the most efficient dependency algorithm but only to have enough constraints to come up with valid programs at the end.

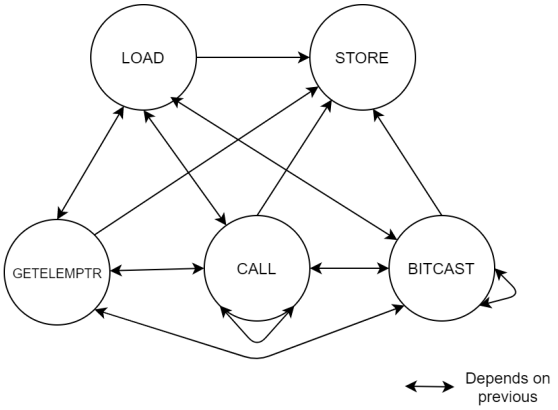


Figure 4.2: Dependencies between memory instructions. This graph defines constraints to reordering of instructions to avoid errors during rescheduling

Instruction	Description
load	Reads the memory
store	Writes to memory
getelemptr	Performs address calculation of a subelement of an aggregate data structure. It does not access memory.
bitcast	Converts a value to a new type without changing any bits
call	Represents a function call. Transfer the controll flow to a specific function.

Table 4.1: Memory Instructions that are considered when creating the additional dependencies.

Algorithm 3: Dependency Graph Construction

```
input  : Instructions set 𝓘 of the basic block
output: Dependency graph D of the basic block
Initialize empty dependence graph D ;
for i in 𝓘 do
    D += node(i);
    /* handle memory related dependencies here                    */
    if i in memory instructions then
        if i is of load type then
            add edges between i and all previous instructions that are not of type
             load;
        else if i is NOT of type store then
            if i is of getelemptr type then
                add edges between i and all previous instructions that are not of
                 type getelemptr;
            else
                add edges between i and all previous instructions;
/* handle instructions dependencies here                          */
for all combination of instructions pairs (I1, I2) in 𝓘 do
    if I1 has I2 in its operands then
        add edge I1 and I2;
    else if I2 has I1 in its operands then
        add edge between I2 and I1;
```

This algorithm will be used to build a dependency graph for each basic block of the LLVM code. Using these graphs, the environment has everything needed to find the schedulable instructions to suggest valid choices to the agent.

4.3 Schedulability of Instructions

As highlighted in the "Environment High Level Algorithm" (Algorithm 1) showing schedulable instructions is an iterative process. The environment needs to walk through the original code block by block and, at each step, determine the set of instructions that are schedulable. This process is summarized by Algorithm 4. First, it needs to keep track of which block is currently being rescheduled. With this information, the environment can retrieve the corresponding dependency graph. The schedulability of an instruction is decided based on 2 criteria:

1. The dependencies relations: To be considered schedulable all the dependencies must have already been scheduled. To implement this, we use a copy of the dependency

graph. A node (i.e instruction) is removed when it is scheduled by the agent. If a node does have an in-degree of 0, it is considered dependent free. By using a copy of the original, the environment does not need to recompute the graph after each episode, saving some precious time.

2. The cases of terminal and phi instructions: These instructions need to be handled with particular attention. Indeed, phi instructions, which are used to select values based on the block predecessor, must always stay at the start of a block. To that end, if the set of schedulable instructions defined by the first rule above contains phi instructions, all non-phi instructions are removed from the set. On the other hand, terminal instructions must stay at the end of a block. To enforce this, if there is a terminal instruction in the set of schedulable instructions with other non-terminal instructions in it, the terminal instruction is removed.

Algorithm 4: Instructions Schedulability of one Basic Block

input : Dependency graph D of the basic block
output: Schedulable instructions set $\mathcal{S}$
G' = copy of the block dependency graph D ;
initialize empty schedulable instructions set $\mathcal{S}$;
`/* Check dependencies relations */`
for *i in* $\mathcal{I}$ **do**
  if *G'.in_ degree(i) = 0* **then**
    add *i* to $\mathcal{S}$;
`/* Check for terminal and phi instructions */`
if *phi instructions and non-phi instructions in* $\mathcal{S}$ **then**
  remove all non-phi from $\mathcal{S}$;
else if *terminal instruction and non-terminal instruction in* $\mathcal{S}$ **then**
  remove terminal instruction from $\mathcal{S}$;

Using this method, we make sure our environment can only show valid instructions to the agent, avoiding any chance of breaking the original code behavior. In the next section, we will explain how the environment can rebuild a new program that can be compiled.

4.4 Creation of the Rescheduled Program

During the instruction selection process, when reordering the instruction of a program, our environment is dealing with some sets of instructions. By simply combining theses sets it does not lead to the creation of a correct program that can be compiled. By a valid program, we mean here an LLVM-IR ".ll" file that can be passed to a clang compiler to produce an executable. Indeed, for the rescheduling part, our environment only needs to manage instructions. However, to recreate the complete program, we need to include

function declaration headers, metadata, labels for jumps, and all the other components that make the compilation possible.

To recreate a valid LLVM program, we first considered the use of the IR builder tools of the llvmlite library. These tools allow creating a complete LLVM-IR by creating modules, functions, and basic blocks directly in Python. However, these tools were conceived to generate code from scratch, in our case, we want to modify an existing code by only changing the order of the instructions. This is not possible using these tools. We still considered using it to fully recreate a new LLVM file by parsing some elements of the original one but this would have been very complicated. Instead, we decided to recreate the new LLVM file by using a copy of the original one and to replace some parts of it by the new sequence of instructions. This process is summarized by Figure 4.3.

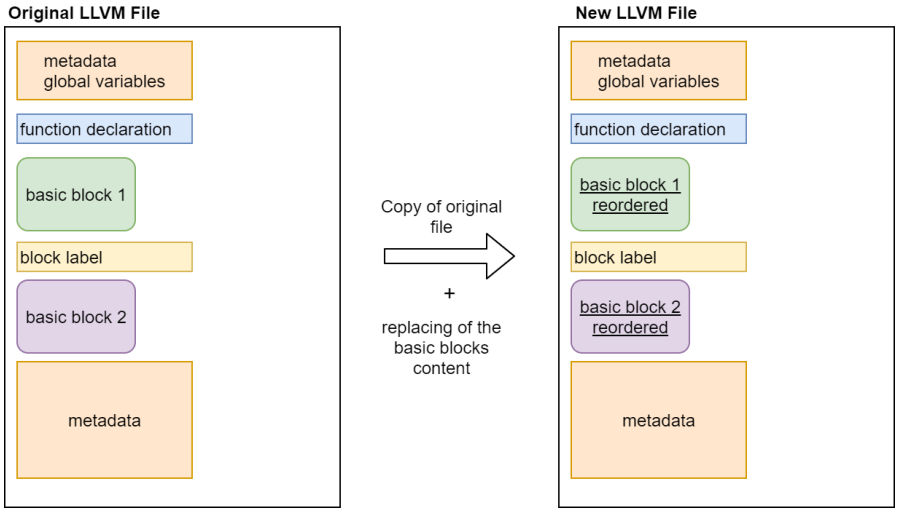


Figure 4.3: Creation of the rescheduled program: High level Procedure

By copying the original file it greatly simplifies the file creation procedure as it allows to only focus on block instructions. All of the other elements (metadata, global variables, labels, etc) do not need to be modified. To replace a basic block to its new version (e.g. the green rectangle in Figure 4.3) we decomposed the process in 2 steps:

- The replacement of the block content: we replaced, in the copied version of the LLVM original code, the content of the basic block by the new sequences of instructions generated by the agent. To find the place were the update needs to be made, we first find the line indexes of the start and end of the block we need to update. This search is done using the original block (not the reordered version !). This search is done using lower and upper boundaries to avoid incorrect matching (e.g. if two exact same instructions are present in the code). When the line indexes of the part to be replaced is found, we can update it with the new sequence of instructions.

- When the first step is done, we need to rename variables names. Indeed, when a LLVM file is generated from a c file all its variables names are integers. Surprisingly, clang does not allow unordered variables: %3 cannot be placed before %2. For these reasons, we need to rename them as the reordering of instruction naturally causes these problems to occur. For the renaming to work, it needs to be done in two steps. First, all the variable names are renamed to dummies that do not exist in the program. Only after this, the dummies are renamed to the correct name (i.e so that variable names are ordered). This is done to avoid overwriting the previous renamings as this step is performed sequentially.

Using this procedure, we are able to obtain at the end of an episode a new program than can be compiled and executed to produce a reward for the agent.

4.5 Measure of the Program Execution Time

The environment emits a reward at the end of each episode. As the goal of the environment is to optimize the program speed, the reward function will naturally be constructed using the run time of the new LLVM program produced by our agent.

To be able to be executed the program is compiled using clang. To make sure the work of the agent is not destroyed during the compilation, some flags are passed as input to clang to avoid the reordering of instructions during the optimization passes. Using the resulting executable, the program can now be timed. However, measuring the execution time of a program is not that straightforward.

Indeed, the execution time not only depends on the code executed but also on the state of the machine it is executed on. For example, the state of the caches, use of the memory and CPU cores are all factors that impact the run time. For this reason we will explore different ways of timing a program execution by testing two different metrics, user run time in seconds and CPU cycles, and aggregation method (min and max). Chapter 5 is dedicated to this part, presenting results of the different methods we explored.

4.6 The Gym Environment

Now that we have seen how all the details specific to LLVM are handled, we will present the more general characteristics of our environment. It has been designed to follow the Gym interface so that it can easily be used by already implemented methods.

4.6.1 State and Action Spaces

The first element we designed was the action space. We decided that our agent will be able to take 2 actions. Either add the instruction shown by the environment to the

program in construction or decide to not schedule it. This is implemented by a discrete action space of 2 values: 0 meaning to schedule the instruction and 1 to not schedule.

The second element is the state space. We have to decide what are the elements that our agent will observe to take his decision. Figure 4.4 shows the state composition.

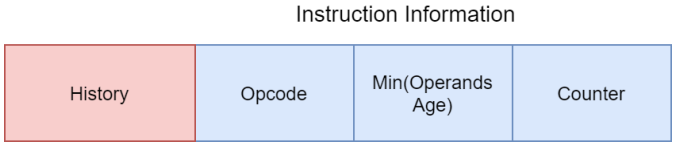


Figure 4.4: State composition. It can be decomposed into 2 components: the history and instruction information

The first and most straightforward element is, of course, the instruction that environment propose to schedule. To represent this instruction we used 3 elements:

1. The instruction opcode where each instruction opcode managed by the environment is translated to an integer.
2. The minimum age of the operands. By the age of the operand, we mean the number of instructions that have been placed in the program between the operand and the instruction. If an instruction has several operands we aggregate the ages by taking the minimum over all the operands. This information could be useful as it may be a good idea to not schedule an instruction just directly after that its operand has been placed into the program to avoid "stalls" (cf. Section 1.2.3).
3. A counter to keep track of the number of times an instruction has been shown to the agent. If an instruction is ignored more than 4 times by the agent, it will be directly placed into the new program. This is done to avoid cases where the agent always refuses to schedule the instruction, causing an episode to never end.

By combining these 2 elements, we are able to incorporate in the state some information about the type and content of the instruction.

Additionally to the instruction information, we are also adding a history of the last 2 instructions that have been placed into the program in construction to incorporate information about the order of instructions into the observation. The history is reset at the beginning of each basic block.

As the state will be the input of neural networks to learn a policy, it is important that the scale and ranges of the values composing it are meaningful. We are using numbers to represent opcodes but these are categorical value. Indeed, the value "3" representing a "mul" instruction is not more important than a "1" representing an "add".

Memory Access Instructions	Description
load	Reads the memory
store	Writes the memory
call	Represents a function call. Transfer the controll flow to a specific function
Arithmetic Instructions	
fadd	Addition of two floating-point values
fsub	Subtraction of two floating-point values
fmul	Multiplication of two floating-point values
fdiv	Division of two floating-point values
Conversion Instructions	
bitcast	Converts a value to a new type without changing any bits
fpext	Extends a floating-point value to a larger floating-point value
sitofp	Converts a value to its corresponding floating-point value
fptrunc	Truncates a value to another type

Table 4.2: Set of instructions recognized by the environment

To avoid problems related to the ordered relationship of these values, we decided to "one-hot" encode them. This encoding works by associating a binary representation with each value. The opcode integer is transformed into an n size vector were each of the n element represents one possible value of the feature (Figure 4.5). In our case, as we have 11 instructions recognized by the environment (Table 4.2) and 1 entry for all others instructions, each opcode is represented by a 12 elements vector. We tried to include different kind of instructions and select the ones we believe can have an impact on the program performances. All the other elements of the state are also one-hot encoded to only have binary values in the state, avoiding ranges discrepancies.

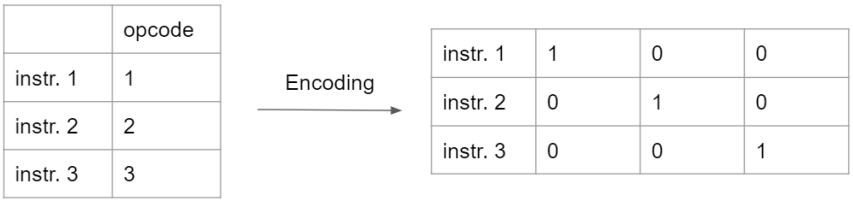


Figure 4.5: one-hot encoding, each instruction opcode is associated with one binary representation.

4.6.2 Reward Function

The reward is computed at the end of each episode when all of the instructions have been scheduled. As we want to encourage the agent to produce the fastest program we want it to be inversely proportional to the program efficiency. The first reward function we tried was:

$$r = -performance(rescheduled\_program) * scaler \tag{4.1}$$

The scaler is there to rescale the run time to avoid really small rewards if the run time is really fast. However, the problem with this function is that the agent will receive only negative rewards which will lead to the agent never choosing to schedule any instructions. To tackle this problem we modified the function by adding the original program performance into the equation.

$$r = \frac{cycles(original\_program) - cycles(rescheduled\_program)}{cycles(original\_program)} * 10 \tag{4.2}$$

$cycles(program)$ represents the number of CPU cycles it took to run. The multiplication by 10 can be viewed as the scaler comparing it to the first reward function but its value can remain fixed and does not have to be adapted for a specific program. Using this reward function, the agent will be rewarded to produce a more efficient program than the original one.

With the reward function, actions, and state definitions we almost have all the elements necessary to train an agent in our environment. The only missing element is the step function (i.e the transition function) that decides how we do move from one state to a new one.

4.6.3 Transition Function

To produce a new observation, the environment first updates the current instruction counter information. Then, if the agent has chosen to schedule the current instruction, it updates the history and a check is done to verify if an episode has ended, leading to the computation of a non-zero reward and a "none" state is returned. Finally, if the episode is not done, a new instruction is selected randomly and a new state is created independently of the action selected by the agent.

Algorithm 5: Step Function

```
input  : action
output: new_state
reward = 0 ;
update current instruction counter ;
if action = O then
    update history;
    if done then
        reward = reward_function(run_time);
        new_state = None;
        exit;
new_instruction = random(schedulables_instructions);
new_state = onehot(history, new_instruction);
```

4.7 Remote Time Measurement

As we mention in the previous section, the measure of a program run time is challenging and is a crucial element for our environment. For this reason, when evaluating a program created by an agent, it is important to try to limit the number of other tasks the Operating System needs to handle. It is common to use an HPC cluster to do experiments that need a long time to run to benefit from more computational resources and avoid the solicitation of our personal computer. However, in our case, we faced one problem with the use of this kind of cluster. If some other tasks are running on the same cluster node, it would make the run time measures even more unstable, leading to even more noisy rewards.

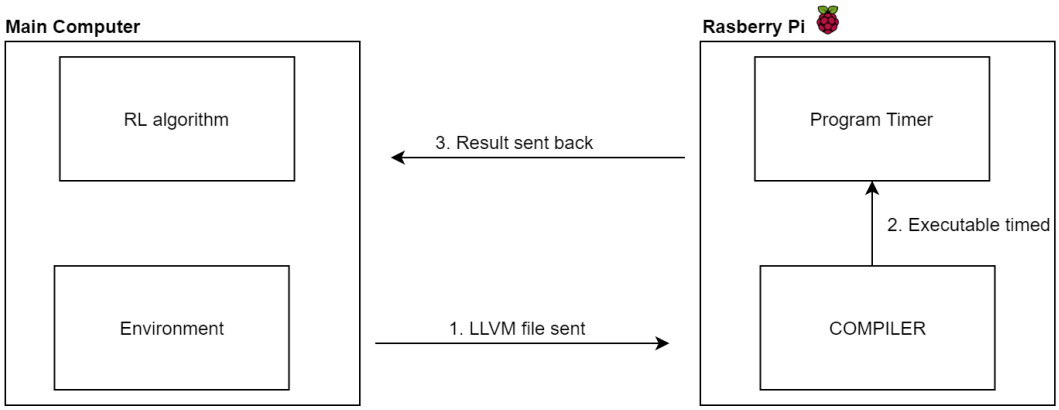


Figure 4.6: Communication process between the Raspberry PI and the main computer.

The solution we propose is the possibility of evaluating a program on a remote de-

vice where we could limit and control the background tasks running at the same time. We chose to use a Raspberry Pi as a remote device as it is a quite cheap solution and easy to setup. Moreover, Raspberry Pis are equipped with much simpler CPU's (Raspberry Pi 3b+ has a Quad-Core 1.2GHz Broadcom BCM2837 64bit CPU) than modern computers which make the measure of a more stable run time easier. Embedded microprocessors, such as the one of the Raspberry Pi, have far less advanced instruction scheduling algorithms than Intel and AMD microprocessors, as found in laptops and desktops. Because there is no way to disable a microprocessor's scheduling algorithm, precisely measuring the impact of our environment, without having its result tampered by what (possibly-suboptimal) choices the processor makes, requires a microprocessor with simple and discreet scheduling algorithms.

When using the Raspberry Pi, it is only used to compile and execute a program. The learning algorithm is still handled by the main computer (e.g. my personal computer). The Raspberry Pi and the main computer are communicating using ssh. The process can be decomposed into 3 steps illustrated in Figure 4.6.:

1. At then end of an episode, the LLVM file produced by the agent is copied on the Raspberry Pi using the Secure Copy Protocol (SCP).

2. The LLVM file is compiled and timed on the Raspberry Pi to obtain the run time measure.

3. The results are sent back to the main computational unit to be used by the learning algorithm.

Note that the process described below would have work with any other devices using a Linux operating system. In the next chapter, we will highlight the challenges that were faced to obtain reliable performance measures for our environment.

Chapter 5

Empirical validation of the environment

Before trying to apply Reinforcement Learning techniques to our environment, we wanted to see how it can affect the performances of a program by reordering LLVM instructions randomly. By randomly, we mean without any learning but the resulting program still needs to be valid (i.e. returning the same result as the original program). By doing this we want to assess if our environment is able to capture performance differences by producing different versions of the same program. This task of evaluating our environment is challenging as the results depend on:

1. The program the environment is being evaluated on. Indeed some programs might bee more sensitive to instruction scheduling than others depending on the instructions they contain.

2. The hardware the environment is being evaluated on. The size of the memory or the CPU architecture are factors that can influence program performance and influence rescheduling.

3. The way the run time measurement is done by the environment: how many times a program is run and what aggregation method is used are the 2 main questions we will assess here.

We conducted a set of experiments to emphasize the challenges we are facing when trying to measure the performance of a program. We will use a c program that computes the pi number by probabilities. This is a small program that has the advantage of being quite fast so we can run it many times (we will run it up to 1000 times per measurements) by keeping the overall time of an experiment very reasonable (less than an hour). Additionally, this program is quite reschedulable. On average, 3.5 instructions can be scheduled at each step.

Different tools exist in python to measure the execution time of a program. In the

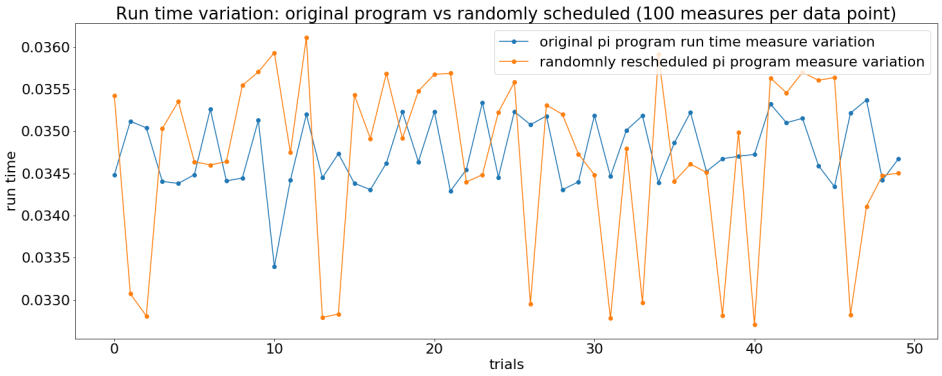


Figure 5.1: Comparison of the variation of the run time measurements in seconds of randomly rescheduled programs versus the original program. Evaluating the same program (the original one) lead to more stable results but they are still noisy.

first place, we used the resource module[1] in combination with the subprocess[2] one. The last one allows us to start running the program independently of the environment by creating a new process. By doing this, if a crash occurs during the execution, it would not cause our environment to crash as well. The resource module allows us to time precisely the process of running the new LLVM code to get the user CPU time used. This measure is well suited to benchmark the performance of a program as it will only measure the time the CPU is spending executing the code itself omitting other processes and time spent in kernel mode used for I/O operations notably.

We evaluated the environment on a laptop with an Intel® Core™ i5-5200U CPU @ 2.20GHz × 4 running Ubuntu 18.04. During the evaluations, we tried to limit the number of tasks running by closing all the other programs.

The first set of results, presented in Figure 5.1 shows the run time difference between the measures obtained when running the same program versus the measures obtain when running a different program across 50 trials. For each data point (i.e. run-time measure), the program was run 100 times and the run times were aggregated using the Minimum operation as it leads to the most stable measurement in this case (Figure 5.2). As expected, we observe that the measures obtained when running the same program over multiple trials are more stable than the ones obtained with different rescheduled programs. However, in some cases, we do not obtain the expected results (i.e original program more stable than rescheduled ones), and this highlights the difficulty of finding the best run time measure.

An example of an unexpected case is shown in Figure 5.3. In this experiment, we

[1] https://docs.python.org/3/library/resource.html
[2] https://docs.python.org/3/library/subprocess.html

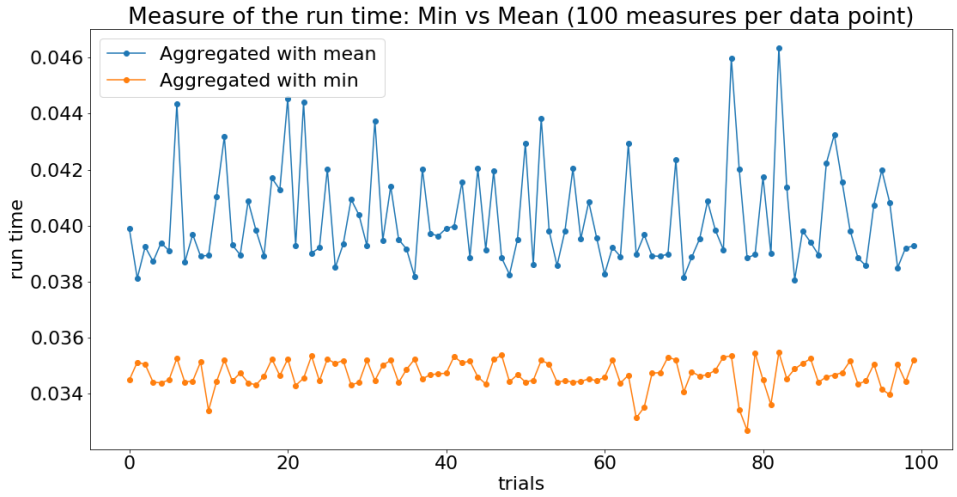


Figure 5.2: Min versus Mean to aggregate run time measurements in seconds of the same program using 100 measures per data points. Using the minimum of the 100 measures leads to a more stable result.

wanted to evaluate how the increase in the number of runs affect our run time measures. By increasing the number of runs from 100 per trial to 1000 per trial. We observe that using the minimum value, in this case, leads to unexpected behavior. Indeed, the run time measures of the original program are more variable than the randomly rescheduled programs (Figure 5.3a). On the other hand, when using the mean over the measurements we come back to the expected case where the randomly scheduled programs are more variable (Figure 5.3b).

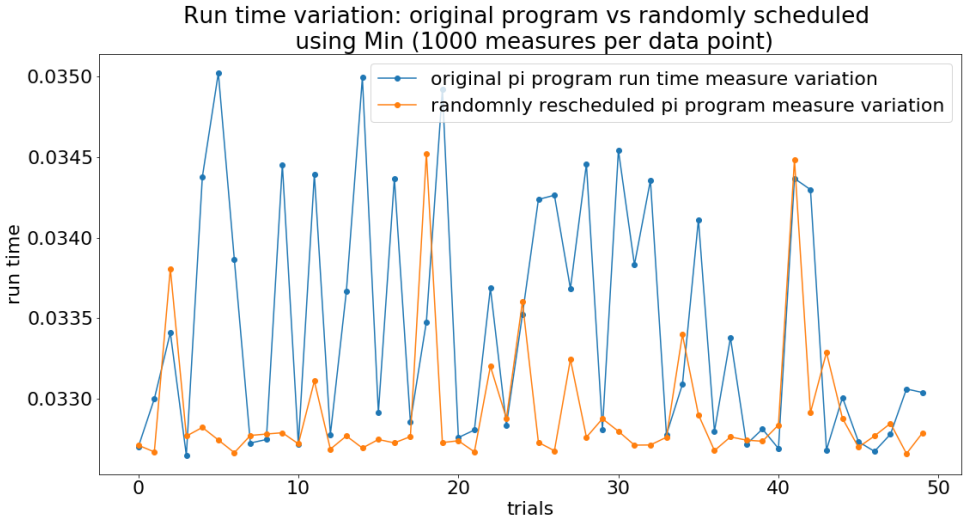
Next, we wanted to investigate how the run time measurements are affected when the program is run on a Raspberry Pi. For this experiment, we used a Raspberry Pi 3b+ equipped with a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU running Ubuntu server 18.04.

When running on this device the run time of a program is naturally higher as the CPU is much less powerful. The slower run time can be viewed as both an advantage and a disadvantage at the same time. Indeed, a slower run time means it is easier to measure since:

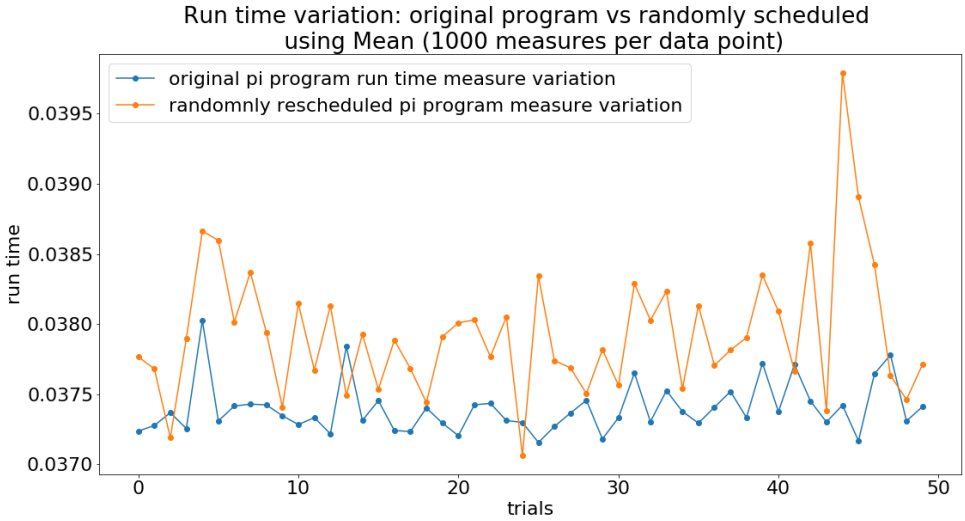
1. The measure is less influenced by noisy signals. If the Operating System decides to perform an action when a 50 ms program is running this would have more consequences than if it was a 2 minutes program running.

2. It requires less precision to measure a longer program.

but on the other hand, a slower run time means a higher episode length (in terms of

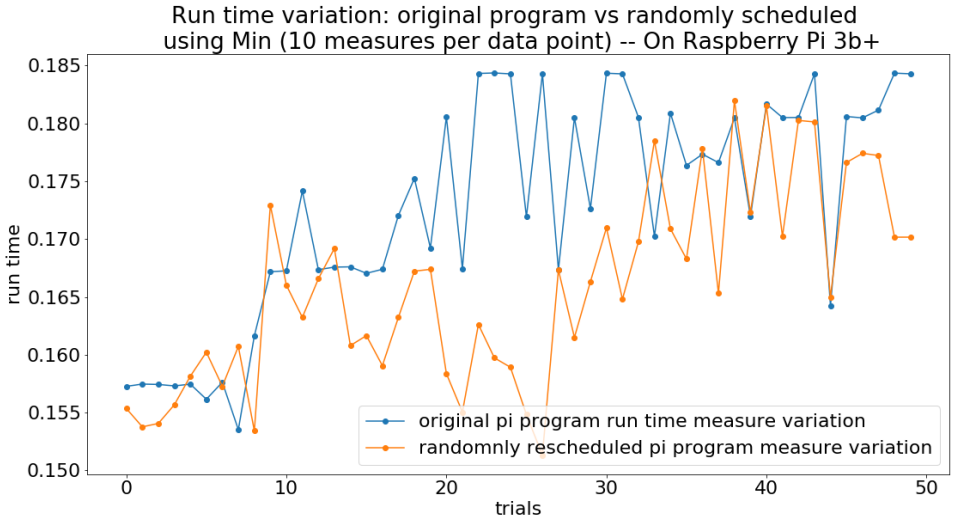


(a)

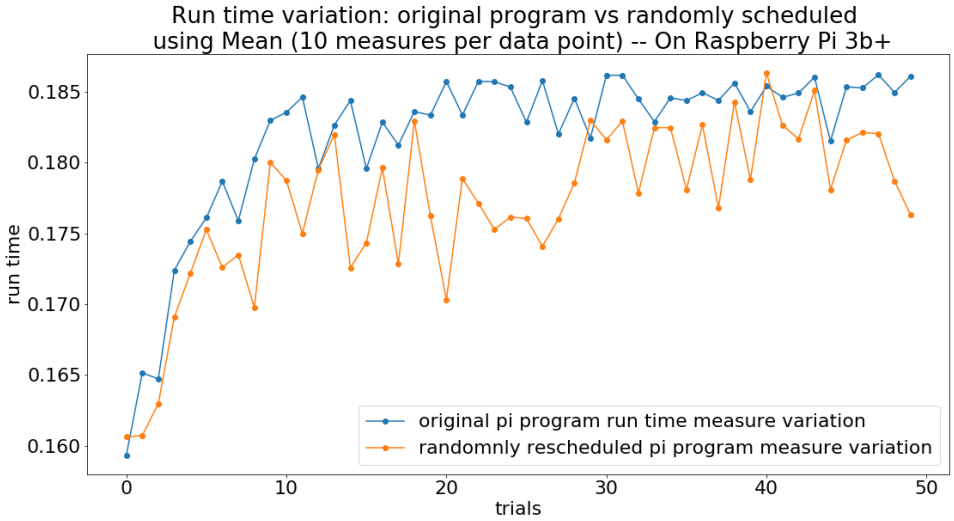


(b)

Figure 5.3: Usage of Min (a) versus Mean (b) for run time measurements in seconds aggregated on 1000 runs. Using the the min is not a good idea in this case as the measures of the original program is more variable than the measures obtained on different random programs.



(a)



(b)

Figure 5.4: Usage of Min (a) versus Mean (b) for run time measurements in seconds aggregated on 10 runs using a Raspberry Pi. Using the mean gives more stable results. A slow down of the Pi is observed after 8 to 10 measures.

time) and so longer training time. In our case, as we are in an extremely sparse setting (one reward per episode), we need to keep the episode time as short as possible to be able to perform as many episodes as possible during training. To keep the time of an episode below what we consider reasonably low ($\geq$ 2 seconds) we could afford to run the program 10 times.

The results comparing the run time variations of the original program versus the randomly rescheduled ones are illustrated in Figure 5.4. The first observation we can make is that the pi tends to slow down after the first 8 to 10 first measures. The second one is that using the mean leads to more stable results (see Figure 5.4b).

Finally, we wanted to try a different performance measure. Until this point, we are using the run time to measure the efficiency of a program but other measures exist. The one we tried is the "CPU cycles" measure obtained using the *perf*[3] tool on Linux. It allows us to measure the time of a program in term of CPU cycles. Using this new measure leads to very interesting results as this leads to a very stable curve when timing the same program multiple times while still having very fluctuating measures when evaluating randomly scheduled programs by doing only 10 runs per measure and taking the minimum. The results are shown in Figure 5.5. The better stability of this measure can be explained by the fact that the CPU can count the exact number of cycles a program is running as it has a counter dedicated to this. If it has to stop executing the program for any reason, it will stop updating this counter until it starts running it again.

To conclude this section, we have shown that our environment could have an impact on program efficiency both in terms of execution time and CPU cycles. Moreover, we demonstrated that the measures used, the hardware and operations used to aggregate the results (i.e minimum or average) have an impact on the quality of the results obtained and so need to be chosen carefully. As using CPU cycles on the Raspberry Pi lead to the more stable measurements we decided to use this technique to evaluate our environment using RL methods.

[3]https://www.man7.org/linux/man-pages/man1/perf.1.html

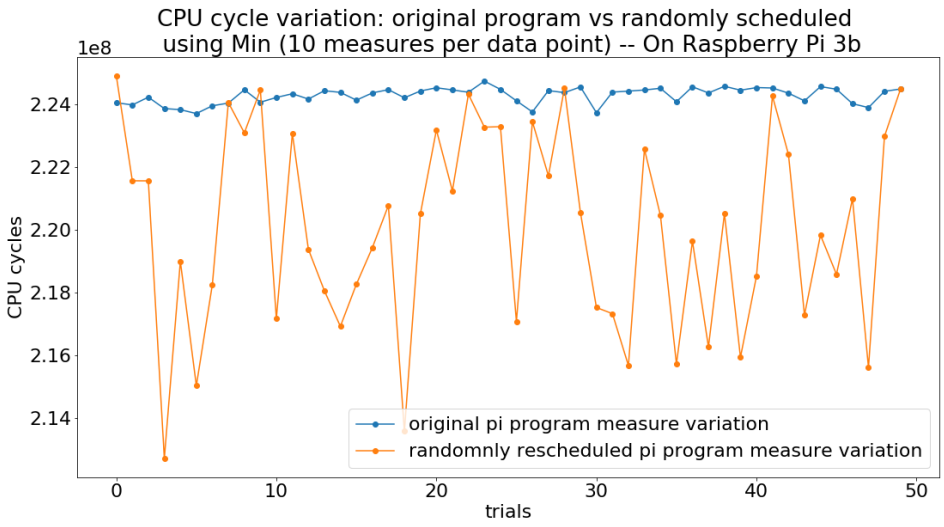


Figure 5.5: Comparison of the variation of the CPU cycles measurements of randomly rescheduled programs versus the original program using a Raspberry Pi. Using CPU cycles leads to stable measurements on the same program.

Chapter 6

Reinforcement Learning Challenges and Perspectives

The final matter we need to investigate in this master thesis is how Reinforcement Learning methods behave when trying to learn in our environment. To do so, we used Stable Baseline[1], a set of publicly available implementations of RL algorithms based on the openAI baselines. The evaluations were performed using the same program used to evaluate performance measures of a program in the last chapter.

6.1 Experiments

First, we evaluated our environment by using PPO. Indeed, policy optimizations methods tend to work best in a partially observable environment. Moreover, the PPO implementation provided by Stable Baselines supports the use of recurrent policies using LSTMs which as discussed in section 2.2 should help an agent learning in a partially observable environment. The "n_step" parameter of PPO, controlling the number of steps to run each environment per update, was set to 1000 (5 times the size of an episode) to avoid making updates with only steps containing no information (i.e. 0 as a reward). The PPO agents were trained for 1e6 timesteps. we repeated the experiment 5 times to average the results. Since a POMDP is harder for an agent to learn, we considered increasing the size of the instruction history observed (cf. Section 4.6.1) in order to improve the observability of our environment. We tried different sizes (2, 4, 6 and 8).

Figure 6.1 shows the comparison between a history of size 2 and 8 as well as the performance of a random agent (i.e. producing random programs). We see that both PPO agents are very unstable. Increasing the history size increase slightly the average which could indicate that the agent struggles with the partial observability of the environment returned but we cannot conclude anything for certain as the 2 standard deviations regions are mostly overlapping. The instability of the agent may also come from the early

[1] https://github.com/hill-a/stable-baselines

rewards it receives

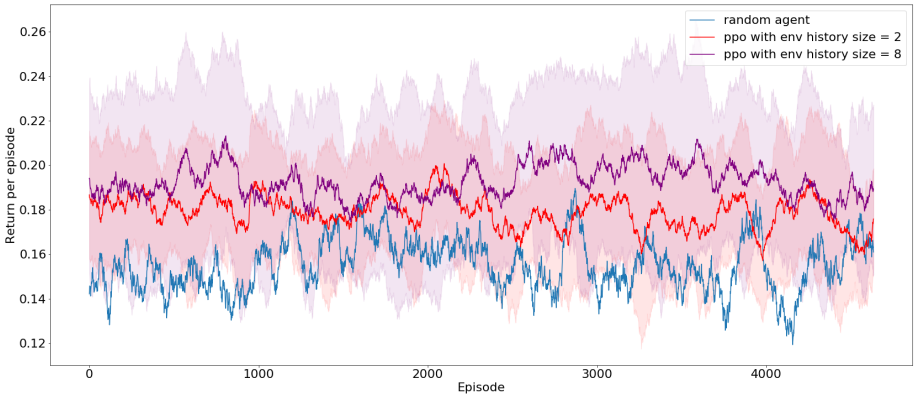


Figure 6.1: Both PPO agents, using a history of size 2 or 8, are unstable. Using a more observable environment seems to slightly help the agent but we cannot conclude anything due to the overlapping of the standard deviations regions.

We could think that one potential source of this unstable behavior may come from the reward signal itself. That could explain the difficulties for the agent to learn a stable policy. However as already shown in Figure 5.5 and confirmed by running an agent that always outputs the same program for 5000 episodes (Figure 6.2) we show that the reward signal is stable and contains low noise levels so that the instability cannot be assigned to the reward signal.

Another source of instability may come from the neural network used as a policy approximator. With the aim to stabilize the learning process, we investigated how the decrease of the learning rate of the methods (i.e. by how much the weights are updated at each optimization step) affects the agent. For this, we tried 2 different learning rates, 1e-3, and 1e-6. However, as shown in Figure 6.3 we did not see any impact on the learning curves.

DQN was also tested in our environment even if we expected it, in theory, to be worse than PPO as q-values based algorithms tend to be less suitable to POMDP. Also due to the sparsity in our problem setting the replay mechanism is less efficient as most of the transitions stored in it give no learning signal as they give a reward of zero to the agent. As expected we did not manage to obtain better results. We did also try to run A2C and TRPO to see if we could get a stable learning process but none of the 2 attempts were successful.

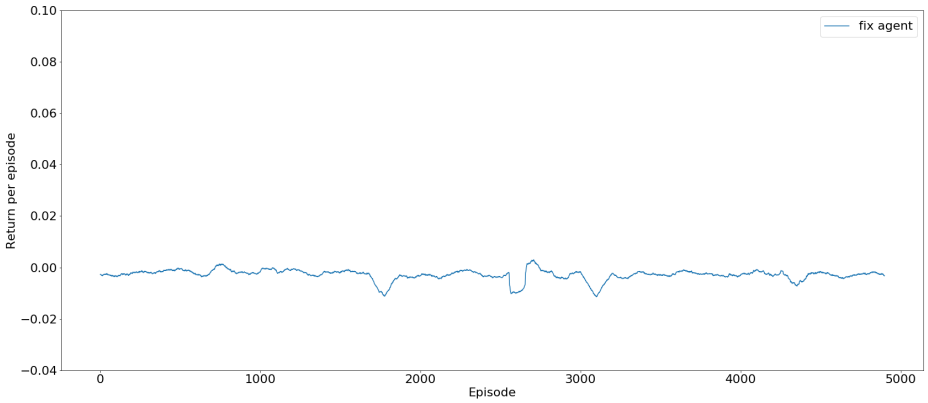


Figure 6.2: Performance evaluation of a fixed agent doing always the same action in the same order. As this leads to a very stable reward, we can conclude that there is a negligible amount of noise coming from the reward function.

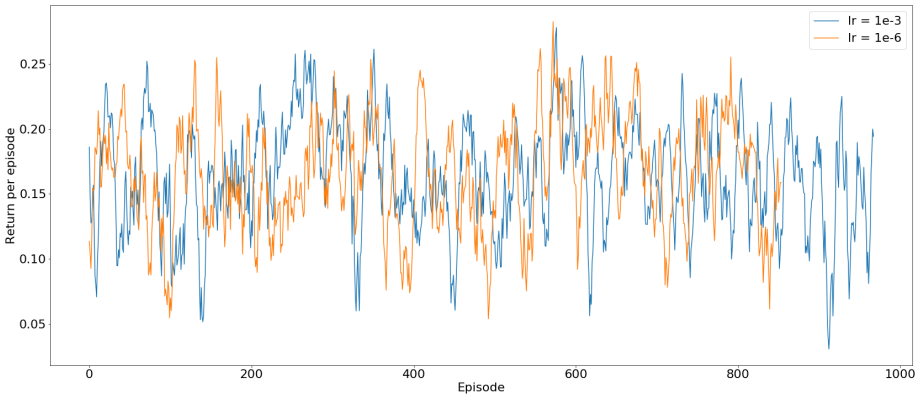


Figure 6.3: Comparing 2 different learning rates, 1e-3 and 1e-6, it does not seem that a smaller learning rate improves the stability of the agent.

6.2 Discussion

6.2.1 Results

This master thesis proposes a new Reinforcement Learning environment that could be used to improve the performance of a program by automatically rescheduling machine instructions.

Environment Implementation

Implementing a functional gym environment capable of rescheduling LLVM machine instructions was the main goal of this thesis. We faced 2 main challenges during this phase. First finding the instructions dependencies constraints was not trivial as we wanted to obtain a valid program at the end without having too many constraints allowing a sufficient level of freedom to the agent to impact the program performances. Second, the process of rebuilding a compatible program containing not only the instructions but also the meta-data and constants needed for a program to work had to be fully implemented by us as the currently available solution to build an LLVM program was not suited to our problem. Debugging this program reconstruction process was quite difficult as the clang compiler does not always provide the most explicit error messages.

Environment Evaluation

Having a functional environment was the first step, but we had to make sure it was capable of influencing the performances and produce a valuable learning signal. For this we evaluated 2 different performance measurements, the user run time and the number of CPU cycles. We also evaluated the influence of the number of runs that needed to be done to obtain a stable measurement and the effect of taking the mean or the minimum of the different runs. To stabilize the performance measures even more, and as a consequence the reward, we implemented a system to execute a program on a remote device (a Raspberry Pi).

Reinforcement Learning Algorithm

We evaluated our environment with PPO (using a recurrent policy network), TRPO, A2C, and DQN. We did not manage to obtain stable policies with any of the algorithms. As we believe PPO is the most suited method among those, we also evaluated it by varying the observability of the environment as well as the learning rate of the policy network. We believe that increasing the number of learning steps may lead to better performances but we could not do it due to computational limitations (we only had one Raspberry Pi).

6.2.2 Future Work

To obtain better performances with Reinforcement Learning methods several solutions could be tried and combined.

First, to tackle the partial observability of the problem it could be useful to incorporate new information to the observations. The good news is that this could be easily done, requiring only some small code modifications. Another possible update on the environment that we believe could improve an agent learning process is the use of a queue to manage the schedulable instructions that have not been scheduled yet. Instead of choosing randomly an instruction among them as we do now, we could always pick

the first instructions of the queue and if the agent decides to not schedule it reinserts it at the end of the queue. This could decrease the randomness of the transition function.

As we mention in the last chapter we think that increasing the number of learning steps when training an agent could improve the performance. A learning episode for an agent takes time as it requires to run the program multiple times in the end. One of the advantages is that it can be parallelized using multiple workers that generate experiences in parallel. Therefore we could imagine using a cluster of Raspberry Pis connected to the same network to gather many more episodes in the same amount of time.

6.2.3 Source code

All the code in Python 3.6 used to build the environment, train the agents, and do the experiments presented in this thesis is available at `https://github.com/DavidEngelman/thesis`. The code depends on the following Python libraries: scp, paramiko, gym, stable baseline, and tensorflow.

Bibliography

[Sut88] Richard S Sutton. "Learning to predict by the methods of temporal differences". In: *Machine learning* 3.1 (1988), pp. 9–44.

[WS91] Deborah Whitfield and Mary Lou Soffa. "Automatic generation of global optimizers". In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1991, pp. 120–129.

[WD92] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[Lin93] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[KKF97] Akira Koseki, H Komastu, and Yoshiaki Fukazawa. "A method for estimating optimal unrolling times for nested loops". In: *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*. IEEE. 1997, pp. 376–382.

[Cas98] Anthony Rocco Cassandra. "Exact and approximate algorithms for partially observable Markov decision processes". In: (1998).

[KLC98] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial intelligence* 101.1-2 (1998), pp. 99–134.

[Mos+98] J Eliot B Moss et al. "Learning to schedule straight-line code". In: *Advances in Neural Information Processing Systems*. 1998, pp. 929–935.

[Fra99] Christopher W Fraser. "Automatic inference of models for statistical code compression". In: *ACM SIGPLAN Notices* 34.5 (1999), pp. 242–246.

[Sut+00] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[Bak02] Bram Bakker. "Reinforcement learning with long short-term memory". In: *Advances in neural information processing systems*. 2002, pp. 1475–1482.

[MMB02] Amy McGovern, Eliot Moss, and Andrew G Barto. "Building a basic block instruction scheduler with reinforcement learning and rollouts". In: *Machine learning* 49.2-3 (2002), pp. 141–160.

[SM02] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[CM04] John Cavazos and J Eliot B Moss. "Inducing heuristics to decide whether to schedule". In: *ACM SIGPLAN Notices* 39.6 (2004), pp. 183–194.

[Coo+08] Katherine E Coons et al. "Feature selection and policy optimization for distributed instruction placement using reinforcement learning". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2008, pp. 32–42.

[Ver09] Dirk Vermeir. *An introduction to compilers*. Feb. 2009.

[Fur+11] Grigori Fursin et al. "Milepost gcc: Machine learning enabled self-tuning compiler". In: *International journal of parallel programming* 39.3 (2011), pp. 296–327.

[KC12] Sameer Kulkarni and John Cavazos. "Mitigating the compiler optimization phase-ordering problem using machine learning". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2012, pp. 147–162.

[PCA12] Eunjung Park, John Cavazos, and Marco A Alvarez. "Using graph-based program characterization for predictive modeling". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 2012, pp. 196–206.

[Kul+13] Sameer Kulkarni et al. "Automatic construction of inlining heuristics using machine learning". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2013, pp. 1–12.

[Mni+13] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[HS15] Matthew Hausknecht and Peter Stone. "Deep recurrent q-learning for partially observable mdps". In: *2015 AAAI Fall Symposium Series*. 2015.

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[Rog15] Yves Roggeman. *Language de Programmation*. Feb. 2015.

[Sch+15a] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[Sch+15b] John Schulman et al. "Trust region policy optimization". In: *International conference on machine learning*. 2015, pp. 1889–1897.

[Wan+15] Ziyu Wang et al. "Dueling network architectures for deep reinforcement learning". In: *arXiv preprint arXiv:1511.06581* (2015).

[ZS15] Wojciech Zaremba and Ilya Sutskever. "Reinforcement learning neural turing machines-revised". In: *arXiv preprint arXiv:1505.00521* (2015).

[VGS16] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Thirtieth AAAI conference on artificial intelligence*. 2016.

[And+17] Marcin Andrychowicz et al. "Hindsight experience replay". In: *Advances in neural information processing systems*. 2017, pp. 5048–5058.

[Sch+17] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[Zhu+17] Pengfei Zhu et al. "On improving deep reinforcement learning for pomdps". In: *arXiv preprint arXiv:1704.07978* (2017).

[Ash+18] Amir H Ashouri et al. "A survey on compiler autotuning using machine learning". In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–42.

[Arj+19] Jose A Arjona-Medina et al. "Rudder: Return decomposition for delayed rewards". In: *Advances in Neural Information Processing Systems*. 2019, pp. 13544–13555.