SoC-FPGA Monte Carlo Tree Search Processor

Ricardo Rodrigues Carvalho

Thesis to obtain the Master of Science Degree in

**Electrical and Computer Engineering**

Supervisor(s): Prof. Horácio Cláudio De Campos Neto

**Examination Committee**

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Horácio Cláudio De Campos Neto
Member of the Committee: Prof. Nuno Cavaco Gomes Horta

June 2019
Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgments

First and foremost I would like to thank my thesis supervisor Prof. Horácio Neto of the Instituto Superior Técnico, who always steered me in the right direction, for his constant support, availability and knowledge.

I must express also my gratitude to my parents, brother and friends for providing me with unfailing support and continuous encouragement through the process of researching and developing this thesis.
Resumo

O Monte Carlo Tree Search (MCTS) é um método de procura em árvore, em que simulações aleatórias são aplicadas com o intuito de encontrar decisões ótimas num sistema. O objetivo deste trabalho consiste na pesquisa e desenvolvimento de uma arquitetura hardware-software do MCTS numa Soc-FPGA, demonstrada em um subconjunto do jogo de xadrez, onde uma arquitetura de hardware dedicada é desenvolvida de forma a acelerar o algoritmo. O uso de funções heurísticas no MCTS é também explorado e avaliado neste trabalho.

O processador dedicado hardware realza jogadas de xadrez, por dois jogadores, a partir de uma determinada posição inicial. Este processador gera todas a jogadas, avalia-as e selecciona uma, repetindo sucessivamente as ações referidas, para os dois jogadores até que uma condição de paragem se verifique.

A aplicação de heurísticas e de tabelas de finais de jogo durante as etapas de seleção, expansão e simulação do MCTS incrementou a precisão do algoritmo e reduziu o tempo de computação do MCTS.

A arquitetura de hardware-software, desenvolvida e implementada numa Zynq-7010, acelerou o algoritmo até 98 vezes. Permitiu realizar todos os jogos exigidos pela etapa de simulação do MCTS e, se escalada para 10 elementos de processamento, com recurso a um método de paralelização em folha, pode atingir uma aceleração até 668 vezes.

Palavras-chave: Xadrez, MCTS, FPGA, Simulações do MCTS em Hardware
Abstract

Monte Carlo Tree Search (MCTS) is a search method that combines a tree search with random simulations in order to find optimal decisions in a system. The objective of this work was to research and develop a hardware-software architecture in a Soc-FPGA for MCTS, applied to a subset of chess, where a dedicated hardware architecture was designed to accelerate the algorithm simulations. The use of specific heuristic functions on MCTS was also explored and evaluated.

The dedicated hardware processor plays a Chess end game from a given starting position. This processor generates all moves, evaluates them, selects one for the side-to-move and successively repeats these actions until a stop condition is verified.

Chess specific heuristics and 4-piece endgame tables in the selection, expansion and simulation stage of MCTS were used to improve the algorithm accuracy and reduce the computation time of MCTS.

The developed hardware-software architecture, implemented in a Zynq-7010, accelerated the MCTS execution up to 98 times. The architecture is fully scalable and if scaled to 10 processing elements, using a leaf parallelization method, can achieve an acceleration up to 668 times.

Keywords: Chess, MCTS, FPGA, Hardware MCTS simulations
Contents

Declaration ................................................................. iii
Acknowledgments ....................................................... v
Resumo ........................................................................ vii
Abstract ......................................................................... ix
List of Tables ............................................................... xiii
List of Figures .................................................................. xv
Nomenclature ............................................................... xix
Acronyms ........................................................................ xix

1 Introduction ............................................................... 1
  1.1 Motivation ............................................................. 1
  1.2 Work Objectives .................................................. 2
  1.3 Target Platform .................................................... 2
  1.4 Thesis Outline ...................................................... 3

2 Chess and Monte Carlo Tree Search ......................... 5
  2.1 Monte Carlo Techniques ......................................... 6
  2.2 Monte Carlo Tree Search ....................................... 6
    2.2.1 Adding Heuristics to MCTS ............................... 10
  2.3 Chess ................................................................. 11
    2.3.1 Faile Chess Engine ......................................... 13
    2.3.2 Syzygy Endgame Tablebases ............................ 16

3 State of the Art .......................................................... 19
  3.1 MCTS Applications ............................................... 19
    3.1.1 Applications Strategies Analysis ..................... 20
    3.1.2 AlphaGo ....................................................... 22
  3.2 Parallelization ...................................................... 23
    3.2.1 Leaf Parallelization ........................................ 23
    3.2.2 Root Parallelization ....................................... 24
    3.2.3 Tree Parallelization ....................................... 24
    3.2.4 Comparison Between Parallelization Methods ......... 25
List of Tables

1.1 Zynq-7000 All Programmable SoC series logic resources. ................................. 2
2.1 A table beside a figure ......................................................... 8
2.2 Initial Chess board using Faile representation ............................................. 13
2.3 Kings table values ...................................................................... 16
2.4 White pawns table values ............................................................. 16
2.5 Black pawns table values ............................................................. 16
3.1 Winning rate attained on MCTS based AI players ............................................. 21
3.2 The speed of decision-making on the parallelization methods (in ms) [23] ... 25
3.3 Winning percentage and confidence interval between parallelization methods [26] ... 26
4.1 MCTS total losses in 100 000 games played against a random algorithm ........ 34
4.2 MCTS stages execution time depending on iterations and simulations per expansion (in ms) .............................................................. 35
4.3 Moves found by different versions of MCTS ................................................. 43
4.4 MCTS execution time for (500 iterations) ..................................................... 44
4.5 MCTS execution time (10000 iterations) ..................................................... 44
5.1 Data on the input register .................................................................. 46
5.2 Data on the output register .................................................................. 46
5.3 Components latency present on the white evaluation circuit ...................... 53
6.1 Data and addresses offsets of the AXI input registers ................................... 60
6.2 Data on the AXI input register-12 ....................................................... 60
6.3 Data on the AXI output register-12 ....................................................... 60
6.4 Algorithm speed-ups achieved by the Hw/Sw system ............................... 64
6.5 MCTS stages execution time for 1 and 10 simulations per expansion .......... 64
6.6 Performance increase on MCTS using 10 processing elements .................. 66
6.7 MCTS stages execution times and average speedup achieved .................... 66
A.1 First turn and Stockfish best move(s) for each position ............................... 78
List of Figures

2.1 Example of a MCTS tree, applied to tic-tac-toe. ................................................. 6
2.2 Example of the selection stage on a MCTS tree ................................................. 8
2.3 Example of the expansion stage on a MCTS tree ............................................... 8
2.4 Example of the simulation stage on a MCTS tree ............................................... 9
2.5 Example of the backpropagation stage on a MCTS tree ..................................... 9
2.6 Pawn first move. ................................................................................................. 11
2.7 Pawn capture. ..................................................................................................... 11
2.8 En Passant move. ............................................................................................... 11
2.9 Rook move. ....................................................................................................... 12
2.10 Knight move. ................................................................................................. 12
2.11 Bishops move. ................................................................................................. 12
2.12 Queen move. .................................................................................................... 12
2.13 King move. ...................................................................................................... 12
2.14 Short Castle. ..................................................................................................... 12
2.15 Long Castle. ..................................................................................................... 12
2.16 Legal position to Castle. .................................................................................... 12
3.1 Tree structure used in [23] to find wolves candidates (from [23]). ...................... 21
3.2 Main parallelization methods for MCTS (from [26]). .......................................... 23
3.3 The win rate under different time limit (in seconds, from [28]). ......................... 25
3.4 Abstract of Intel Xeon Phi microarchitecture (from [30]). ................................. 27
3.5 Win ratio on TSUBAME 2.0 implementation (from [29]). .................................. 27
3.6 Implemented architecture in [31] (from [31]). ................................................... 28
3.7 Diagram of Solver module (from [31]). .............................................................. 29
3.8 Proposed architecture in [32] (from [32]). .......................................................... 30
4.1 Tic-Tac-Toe Position 1. ......................................................................................... 34
4.2 Tic-Tac-Toe Position 2. ......................................................................................... 34
4.3 Tic-Tac-Toe Position 3. ......................................................................................... 34
4.4 Old (left) and new (right) tree structure ............................................................. 35
4.5 Results of 10 000 simulations in Position 1 for the original and simplified random model. 42
<table>
<thead>
<tr>
<th>Position</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>76</td>
</tr>
<tr>
<td>15</td>
<td>76</td>
</tr>
<tr>
<td>16</td>
<td>76</td>
</tr>
<tr>
<td>17</td>
<td>76</td>
</tr>
<tr>
<td>18</td>
<td>76</td>
</tr>
<tr>
<td>19</td>
<td>77</td>
</tr>
<tr>
<td>20</td>
<td>77</td>
</tr>
<tr>
<td>21</td>
<td>77</td>
</tr>
<tr>
<td>22</td>
<td>77</td>
</tr>
<tr>
<td>23</td>
<td>77</td>
</tr>
<tr>
<td>24</td>
<td>77</td>
</tr>
<tr>
<td>25</td>
<td>77</td>
</tr>
<tr>
<td>26</td>
<td>77</td>
</tr>
<tr>
<td>27</td>
<td>77</td>
</tr>
<tr>
<td>28</td>
<td>77</td>
</tr>
<tr>
<td>29</td>
<td>77</td>
</tr>
<tr>
<td>30</td>
<td>77</td>
</tr>
</tbody>
</table>
Acronyms

**AXI** Advanced eXtensible Interface

**BRAM** Block Random Access Memory

**CPU** Central Processing Unit

**DTM** Distance To Mate

**EWN** Einstein Würfelt Nicht

**FF** Flip-Flop

**FIFO** First In First Out

**FPGA** Field-Programmable Gate Array

**GP** General-Purpose

**GPU** Graphics Processing Unit

**Hw** Hardware

**LSB** Least Significant Bit

**LUT** Lookup Table

**MCTS** Monte Carlo Tree Search

**Mutex** Mutual Exclusion

**OCM** On-Chip Memory

**PE** Processing Element

**PL** Programmable Logic

**PS** Processing System

**PTSP** Physical Travelling Salesman Problem

**PUCT** Polynomial Upper Confidence Trees

**RAM** Random Access Memory
ROM  Read-Only Memory
SoC  System-on-a-chip
Sw  Software
TD  Tag Directory
UART  Universal Asynchronous Receiver-Transmitter
UCT  Upper Confidence Bound for Trees
VPU  Vector Processing Unit
WDL  Win / Draw / Loss
Chapter 1

Introduction

Monte Carlo Tree Search (MCTS) is a search method that combines a tree search with random simulations in order to find optimal decisions in a system. For each iteration of MCTS, a tree policy is used to find the most promising node of the tree in order to expand it. The tree policy attempts to balance the tree exploration and exploitation. A simulation is then run from the expanded node and the search tree updated according to the result. In the simulations, actions are applied according to a default policy, which in the simplest case is to perform random actions, until an ending is reached (for example in a game application, a win, lose or draw). MCTS relies on a massive number of simulations to achieve a good solution, therefore performing these simulations on a dedicated hardware architecture, can significantly improve the efficiency of this algorithm, which is the objective of this work.

In recent years, there has been an increasing interest on MCTS algorithm, which led to the forthcoming of several implementations generating a demand related to computer engineering.

1.1 Motivation

The MCTS had showed a massive potential due to its success in situations where deterministic algorithms had failed, especially when applied in domains with vast search spaces. Nowadays, its emerging success led to many MCTS implementations most of them game-related, showing best results when applied to 2-player games with perfect information. AI Go programs is a good example of a MCTS implementation that shown great results, in which until this moment many other methods had failed.

This search method relies on a large number of simulations, therefore demanding a lot of computational power to achieve output correctness.

In this thesis, a System-on-a-chip (SoC) Field-Programmable Gate Array (FPGA) is used to combine efficient embedded general-purpose processing with dedicated hardware. This way, the most time consuming tasks of the MCTS algorithm can be accelerated in hardware, while the least demanding ones can be executed in software.
1.2 Work Objectives

The objective of this work is to research and develop a hardware-software architecture in a Soc-FPGA for MCTS, applied to a chess endgames.

The use of heuristic functions on MCTS is explored and since the computations of heuristic functions turn the system slower there is going to be, as well, an evaluation of the benefit/cost ratio about the usage of these functions. To evaluate how the heuristic functions affects the algorithm accuracy, the best move chosen for a game position by MCTS is compared against a classic chess search algorithm.

Since the game of Chess requires a complex move generator and evaluation function, the MCTS simulation stage is the most time consuming stage of MCTS. Therefore, a dedicated hardware architecture is developed to play a Chess endgame from a given starting position for both white and black sides. This architecture generates all moves, evaluates them, selects one for the side-to-move and successively repeats these actions until a stop condition is verified. The moves are evaluated accordingly to the implemented heuristic function. A strategy to reduce the computation time of heuristic functions is also applied in the designed architecture.

The hardware-software architecture is evaluated in speed where the spent execution time of the implementation is compared against a software-only approach of the MCTS algorithm.

1.3 Target Platform

The system will be demonstrated using a Zynq-7010 device from the Zynq-7000 SoC-FPGA family. A SoC-FPGA allows the design of systems with tightly coupled software based control, due to the presence of the Processing System (PS) and analytics with hardware-based processing and optimized system interfaces, thanks to the existence of the Programmable Logic (PL).

The PS of the Z-7010 device has a dual-core ARM CortexTM-A9 processor that operates at 650 Mhz. The On-Chip Memory (OCM) has a capacity of 256 KB.

The PL has 17600 Lookup Tables (LUTs) and 35200 Flip-Flops (FFs) and 240 KB of extensible Block Random Access Memory (BRAM) memory. Table 1.1 shows the available PL resources in the smallest and largest device from the Zynq-7000 family.

<table>
<thead>
<tr>
<th>Device</th>
<th>Z-7010 (Smallest)</th>
<th>...</th>
<th>Z-7100 (Largest)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>17600</td>
<td></td>
<td>227400</td>
</tr>
<tr>
<td>FFs</td>
<td>35200</td>
<td>...</td>
<td>554800</td>
</tr>
<tr>
<td>BRAM</td>
<td>240 KB</td>
<td></td>
<td>3020 KB</td>
</tr>
</tbody>
</table>

Table 1.1: Zynq-7000 All Programmable SoC series logic resources.

The connectivity between the PS and the PL will be implemented using the General-Purpose (GP) Advanced eXtensible Interface (AXI) ports.
1.4 Thesis Outline

The thesis is organised in the following chapters:

- Chapter 2 - the MCTS algorithms are reviewed and an introduction to the game of Chess is provided;
- Chapter 3 - existent works on the use of MCTS for specific applications and MCTS parallelization methods are presented and analysed;
- Chapter 4 - the software development of MCTS is described. Also, a specific Chess heuristic and 4-piece endgame tables implemented and evaluated;
- Chapter 5 - the hardware implementation of a processing element capable of playing a Chess endgame from a given starting position is described;
- Chapter 6 - the integration of the developed architecture in a Hardware-Software system is described. The results achieved with the Hardware-Software system are compared with the Software-only approach. A parallelization method using multiple processing elements is also proposed;
- Chapter 7 - a work conclusion is presented along with future work to be done;
Chapter 2

Chess and Monte Carlo Tree Search

Machine versus human playing games have always brought a lot of people's attention. In 1997, the world chess champion Garry Kasparov lost a Chess match against IBM's Deep Blue, in a total of six games, Kasparov lost two, won one and ended three in a draw. Since more than a decade ago, computers have outperformed humans in chess, so much that, in 2006, Prof. James A. Hendler wrote "Computers play chess; Humans play Go" [1], because the methods used in Chess computers did not show good results when applied in Go. A major difference between the two games is the average branching factor, which in Chess is of around 35 moves [2], while in Go is of about 250 moves [3]. This is the main reason why the algorithms used in Chess had shown poor results when tested in Go.

In 2006, there was a revolution in Go computer programs with the successful introduction of Monte Carlo methods in the Crazy Stone program, which won the gold medal at the 2006 Computer Olympiad [4]. Crazy Stone success started a trend of using Monte Carlo methods in AI Go computers. Currently most Go programs apply MCTS with several enhancements, like MoGo [5], MyGoFriend [6] and MANGO [6].

MCTS is a heuristic search algorithm that analyses how different inputs affects a certain system through several simulations performed on it. So, MCTS is capable of finding which is the input that maximizes the system output.

Even with the use of MCTS in Go programs, these were not able to outperform humans. Until, the state of art and a big landmark in Go programs, Google DeepMind's AlphaGo [7], defeated a Go world champion, in March 2016. AlphaGo was able to stand out from all others Go programs thanks to a massive computational power and the use of deep neural networks to perform selection and simulation on MCTS [8][7]. After the success of MCTS applied in Go AI programs, there are now some applications of this algorithm applied in Chess. For example, Komodo MCTS which is a Chess program that combines both MCTS and a classic chess algorithm [9]. AlphaZero, the AlphaGo successor, is also a program developed to play several games including chess where a algorithm based on MCTS was used. On AlphaZero, the rollouts used by MCTS were replaced by a evaluation neural network [8].

The following sections review MCTS algorithms and provide an introduction to the game of Chess.
2.1 Monte Carlo Techniques

Monte Carlo method uses several random samplings on a certain system to learn about its outputs, that is, by simulating that system, many times, with random inputs, it heuristically calculates the outputs probabilities for the inputs used on the system.

It is possible to realize that Monte Carlo method is a powerful tool when applied in deterministic models. In a deterministic model, the outcomes are the result of actions applied on the initial state, which is, if with the same initial state and the same actions applied, the outcome will always be the same. Once random inputs are used and applied numerous times in Monte Carlo method, the output of each input is known.

This method can be applied in nearly every quantitative subject, like physical sciences, engineering, statistics, finances, and computing [10]. The vast areas in which this method can be applied, and its very direct approach, turns Monte Carlo method into a very powerful and flexible tool.

2.2 Monte Carlo Tree Search

MCTS is a regular tree search algorithm, where the Monte Carlo method is applied to expand the tree (see figure 2.1). In this algorithm, every tree node is an outcome state of an action applied to the parent node. Every time a new node is added to the tree, this one is simulated until it is possible to be evaluated. Since many nodes exist in a tree, therefore many simulations are performed, showing how the Monte Carlo method is used through the tree development. Every node keeps the information of how many times it was visited and how many times its state had led to a winning. The number of visits and winnings ("P" and "W" on figure 2.1, respectively) is used by the algorithm to select the new node to be expanded. In the example of 2.1 the branching factor is limited to a maximum value of three, for a purpose of representation, that is each node has a maximum of three children.

Figure 2.1: Example of a MCTS tree, applied to tic-tac-toe.
In MCTS are performed four steps. First, the most promising node is selected using a policy technique. Once selected, the node is expanded and simulated. As last step, the information of the node and of every parent are updated with the simulation result until the tree root is reached. Upper Confidence Bound for Trees (UCT) is the most popular policy technique used in the MCTS to perform the node selection. This selection technique is based on the theory on the multi-armed bandit problem. The multi-armed problem can be exemplified by composing a strategy to maximize the gambler's profit at a slot machine with several levers, where each one may have a different payout. This strategy concerns on what lever to play, how many times should the gambler play each lever and in which order should the levers be played [11]. Therefore, the gambler must balance between exploring each of the levers to determine their value and exploiting the ones he already knows in order to benefit from the high paying levers. In this selection technique the tree nodes are visited and expanded taking into account all the a priori runs of the MCTS, UCT has the following formulation:

\[
    UCT = \frac{w_i}{n_i} + C \sqrt{\frac{\ln(n)}{n_i}}
\]

Where \(w_i\) is the number of wins in child node \(i\), \(n_i\) is the number of times that child node \(i\) has been visited, \(n\) is the number of times that the parent node as been visited and \(C\) is a tunable parameter. However \(\sqrt{2}\) is the theoretically ideal value for \(C\), this value is calculated empirically most of the times in such a way that creates the best balance between exploitation and exploration on the MCTS tree.

The UCT formula provides some balance between the exploitation of nodes with high win rate (first term) and the exploration of nodes with low visits (second term). The constant \(C\) makes possible to adjust the trade-off between exploitation and exploration. UCT performs in order to avoid the algorithm from always choosing to exploit a node with a higher number of wins, instead of exploring a better node with a smaller number of wins.

MCTS algorithm is split into the following four stages:

- **Selection:**

  The algorithm starts by selecting the root and choosing one of its children. The tree will be traversed by successive node selections until a non-existent child is found. Figure 2.2 exemplifies the selection stage, starting from root and ending in a non-existing node (position to be expanded). First, the root child with the biggest UCT value is chosen, which is node 1 (see table 2.1). The same action is performed on node 1. The selection stage reach the ending since a non-existing node was selected. The values present on table 2.1 were computed with the \(C = 1.41\). The underline values shown in this table represent the chosen nodes.

  It is worth to mention that, if more than one node exists with the biggest UCT value (e.g. two brother nodes with equal UCT value, being this value the biggest), might be chosen without any preference.
• Expansion:

A new node is created and inserted in the tree with the new state corresponding to the non-existent child previously selected. In figure 2.3, the new node state is its parent state where a new “x” was applied in a available position.

• Simulation:

On the newly created state random actions (or heuristically-based) are repeatedly applied until a state that is possible to evaluate it as a win or a loss is reached, that is, reaching an end of the game. Figure 2.4 exemplifies a possible simulation that ended in a win for the root player.
• Backpropagation:

After simulation, the new node created is updated with the new winning and visit values. These values are back propagated through all parents, up-to the root. In figure 2.5, all nodes present on the path represented in bold are updated with the new values.

MCTS only needs to know which actions are allowed to perform in a certain state and to perceive when a state is an ending situation, and if so, who won. So, in applications that require hard evaluation functions, MCTS thrives. Variable runtime is also a big advantage of this algorithm. Since MCTS is essentially a running loop of all the four stages, it can be cut off any time. This way it is possible to tune how much time is given to the program to "think". Of course it performs better when more time is
given to run since it can run more simulations, increasing the size of its search tree. Once stopped, [12] describes four methods for choosing the best action:

- Max child - Select the root child with the highest ratio of wins per visits;
- Robust child - Select the most visited root child;
- Max-Robust child - Selects the root child with the highest visits and highest ratio of wins per visits. If none exist, MCTS should continue;
- Secure child - Select the root child which maximizes the policy technique (UCT for example).

Another point to consider is that the tree present on MCTS is reusable. This means that when a new best move is computed and played, all previous computations made by the algorithm for the corresponding node still apply, and the corresponding subtree becomes the new tree.

### 2.2.1 Adding Heuristics to MCTS

The MCTS can be improved by including specific domain knowledge in the tree search. Progressive Bias is an example of a policy technique that gives MCTS a domain knowledge using an heuristic function. To achieve this, a new term is added to the UCT equation, obtaining the following formula:

\[
ProgressiveBias = \frac{w_i}{n_i} + C\sqrt{\frac{\ln(n)}{n_i}} + \frac{H_i}{n_i + 1}
\]

(2.2)

where \(H_i\) is a heuristic function that evaluates the child node \(i\) state. Since typically there already are many heuristic functions used in other non-MC methods. An edge of Progressive Bias is that, these functions can be easily adapted to be used in MCTS [6].

An alternative is to include the heuristics in the UCT exploration and exploitation terms, as AlphaGo [8], that uses the following formulation, which is considered a variant of Polynomial Upper Confidence Trees (PUCT) [8]:

\[
PUCT_{variant} = (1 - \lambda)\frac{w_{v_i}}{n_i} + \lambda\frac{w_{r_i}}{n_i} + C_{puct}P(i)\sqrt{\frac{n}{1 + n_i}}
\]

(2.3)

where \(C_{puct}\) is the exploration parameter, \(P(i)\) is the probability of child \(i\) ending in a win, \(w_{r_i}\) is the number of wins of the child \(i\), \(w_{v_i}\) is the value of an evaluation of the board state and \(\lambda\) is a weighting parameter. To compute \(P(i)\) and the value of \(w_{v_i}\) AlphaGo uses deep neural networks which, in this case, acts as domain-specific knowledge functions.

The usage of heuristic functions improves selection on MCTS, mainly, when few simulations were performed. Since, MCTS relies on many simulations to learn about the application, the heuristic functions usage turns possible to MCTS to perform a more appropriate selection even when it has low information about the application behaviour.
2.3 Chess

The game is played on a chessboard with 64 squares. The board has 8 horizontal and 8 vertical lines, the horizontal ones are called ranks and numbered from 1 to 8, while the vertical ones are called files and are distinguished by letters, ranging from a to h. Chess is a two-player game where each one controls 16 pieces, to differentiate the player there are white and black pieces. The player that controls the white ones plays first. Each game starts with eight pawns, two rooks, two knights, two bishops, one queen and one king for each player. The goal of the game is to capture the other player’s king. When a piece moves onto a square occupied by an opponent’s piece is called capture. Once the piece is captured and taken of the game it can only come back if the player promotes a Pawn (explained further below). A move that leaves the player’s king under attack from another piece is illegal to perform. While the king is under attack, it may move to a different square where it won’t be under attack, the path between both pieces can be blocked or the attacking piece may be captured. If it is impossible to do any of the three defensive movements referenced, the game is ended by “checkmate”. The game may also end in a tie if the player has no legal moves and his king is not under attack (stalemate), if 50 consecutive moves are played without a capture or a pawn move (fifty-move rule) or if the same piece’s position occurs 3 times (threelfold repetition).

Each piece type has a different move pattern, which will be explain below.

In the Pawn first move, it can move 2 or 1 squares to the front (figure 2.6). Once moved, it can only move one square. The Pawn can only capture other piece by moving one square diagonally forward, to the left or to the right (figure 2.7). There is also a special move that can be done called “En Passant”. When the opponent moves one pawn two squares forward making it be side to side with one of the user’s Pawn. On the following turn, the user can move his pawn diagonally. Despite not staying on the opponent’s pawn square, that pawn is captured. Figure 2.8 exemplifies this move, where the player can move his white pawn to H6 capturing the black pawn that occupies the H5 square. When a pawn reaches the opposite side of the board, it may be promoted to: a Rook, a Bishop, a Knight or a Queen.

The Rook can move within a file or a rank any number of squares (figure 2.9), but it can’t go over other pieces.

The Knight is the only piece that can move over other pieces. It as a L shape movement, going 2 squares forward or back and 1 square left or right, it can also 1 square forward or back, and 2 left or
right (figure 2.10).

![Figure 2.9: Rook move.](image1)
![Figure 2.10: Knight move.](image2)
![Figure 2.11: Bishops move.](image3)

The Bishops can only move diagonally (figure 2.11). According to their starting position, one bishop can only move on the white squares, while the other can only move on the black squares.

The Queen can move within files, rank or diagonally, without going through other pieces (figure 2.12).

![Figure 2.12: Queen move.](image4)
![Figure 2.13: King move.](image5)

The King, can only move 1 square in each direction (figure 2.13). There is an exception, called castling. This move consists of moving the King two squares to the left (figure 2.15) or to the right (figure 2.14). If it moves to the left, the rook on its left moves two squares to the right, if the king goes to the right, the rook on its right moves three squares to the left. For castling to be possible both the King and the Rook can’t have been moved before (figure 2.16), there can’t be pieces between them and the squares between them cannot be under attack.

![Figure 2.14: Short Castle.](image6)
![Figure 2.15: Long Castle.](image7)
![Figure 2.16: Legal position to Castle.](image8)
The moves generation rules and stop game conditions are required by MCTS during the expansion and simulation stages. Therefore, the information present on this subsection will be used during the development of a MCTS Hardware-Software application.

2.3.1 Faile Chess Engine

The development of a hardware dedicated architecture to accelerate MCTS is the main focus of this thesis. Therefore, several open source Chess engines were analysed, in order to use their moves generators and evaluators as baselines for this work. The source codes of the Chess engines Faile [13], Ethereal [14], GNU Chess [15], Scorpio [16] and Stockfish [17], open source Chess engines written in C and C++, were inspected. Faile chess engine was the one chosen to be used through the course of this thesis, due to its better structured source code and its isolated move generation and evaluation functions.

This Chess engine uses a 144 integer array to represent a chess board. Table 2.2 exemplifies this board representation method, using the initial game piece positions.

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>9</td>
<td>5</td>
<td>11</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.2: Initial Chess board using Faile representation.

Listing 2.3.1 clarifies each integer value meaning on this representation. The two extra rows around the board, classified as Boarder, are used by Faile during the moves generator.

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Border;</td>
<td>7</td>
<td>White Rook;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>White Pawn;</td>
<td>8</td>
<td>Black Rook;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Black Pawn;</td>
<td>9</td>
<td>White Queen;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>White Knight;</td>
<td>10</td>
<td>Black Queen;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Black Knight;</td>
<td>11</td>
<td>White Bishop;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>White King;</td>
<td>12</td>
<td>Black Bishop;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Black King;</td>
<td>13</td>
<td>No piece.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 2.1: Pieces classification on Faile

Faile generates a move by applying a pre-calculated offset on the piece start position. For example, to move a King one square to the right is applied an offset of 1 to its starting position. There are two
distinct methodologies while generating moves. One for the non-sliding pieces, where all offsets are applied once, generating one possible move for each offset. Other for the sliding pieces, where all offsets are applied in loop until an illegal move or a capture is generated. A move is considered illegal when its target square already contains a player's piece or reached the border.

Faile uses a specific data structure to encode moves (listing 2.2) where ep is if it is an en passant move, from the position of the piece, target the position where it goes, captured the piece captured on the target position, promoted if exists a pawns promotion and castled if is a castling move.

```c
struct {
    bool ep;
    int from, target, captured, promoted, castled;
} move_s;
```

Listing 2.2: Data structure for moves in Faile

Faile has three different evaluation functions for initial (more than 11 pieces), mid (from 5 to 11 pieces) and endgame (less than 5 pieces) situations. The total pieces count on board determine which evaluation function must be used, although Pawns and Kings are not accounted.

Since the endgame definition of Faile fits the Chess subset where MCTS will be demonstrated, the endgame evaluation function of Faile will be further analysed. Algorithm 1 shows how the white side is evaluated in Chess endgames. The black side evaluation is done in the same way but each score value is subtracted from the final score, instead of being added.

The evaluation algorithm contains definitions for specific pieces situations used to increase or decrease the score. These definitions will be clarified on the following lines:

- Backwards Pawn - Is the backwards Pawn when compared with the same colour Pawns on the adjacent files, in case there are no pawns in the adjacent files is also considered a backwards Pawn;
- Isolated Pawn - There are no same colour Pawns on the adjacent files;
- Exposed Pawn - There is not an opponent pawn on the same file;
- Passed Pawn - There are not opponent pawns on the same and adjacent files.
- Open File - There are no pawns of either colour on this file.
- Half-Open File - The player has no pawns on this file, but the opponent does.

Listing 2.3 shows the pieces values used by Faile and tables 2.3, 2.4 and 2.5 display the values given for each occupied position used in the end-game evaluation function.
Algorithm 1 End-game Evaluation in Fable

1: procedure ENDGAME_EVAL(int board[], int player)
2:     score = 0
3:     for every piece on board do
4:         if piece = white pawn then
5:             score += pawn_value + white_pawn_table[piece_position]
6:         if is backwards pawn then
7:             score -= 8
8:         if is isolated pawn then
9:             score -= 5
10:        if exposed pawn then
11:            if is backwards pawn then
12:                score -= 3
13:            if is isolated pawn then
14:                score -= 5
15:        if exists more than one white pawn on the same file then
16:            score -= 3 \times \text{(number of white pawns on file)}
17:        if is a passed pawn then
18:            score -= 3 \times \text{white_pawn_table[piece_position]}
19:            if is isolated pawn then
20:                score -= 18
21:        if piece = white rook then
22:            score += rook_value
23:            if rank = 7 then
24:                score += 12
25:        if is half open then
26:            score += 5
27:        if is open then
28:            score += 3
29:        if piece = white queen then
30:            score += queen_value
31:        if piece = white king then
32:            score += kings_table[piece_position]
33:        if player = black pieces then
34:            score = -score
35:     return score
A move generator is required during the expansion and simulation stages of MCTS. So, the move generator from Faile will serve as baseline to implement one for both these stages. Also, the endgame evaluator from Faile will be used as a starting point to develop a function that will add specific domain knowledge to MCTS.

### 2.3.2 Syzygy Endgame Tablebases

Tablebases are precalculated endgame position evaluators generated by an exhaustive retrograde analysis.

There are Tablebases with different metrics, such as Distance To Mate (DTM) and Win / Draw / Loss (WDL) among others. Considering all metrics, WDL is the ideal one since it has a smaller size than other metrics and contains all the necessary information needed by MCTS. With a WDL Tablebase, the MCTS simulations are possible to be evaluated without reaching a win, draw or loss situation.

The two main existing Tablebases with WDL metrics are Syzygy and Scorpio Bitbases [18]. Both were analyzed, and Syzygy was selected due to its lower size (1.3 MB vs 3.6 MB) for 4-men WDL files. Other reason for selecting Syzygy Tablebases was that Fathom [19], a stand-alone Syzygy probing tool, contains a better structured code therefore easier adapt than Scorpio.

Fathom uses a specific data structure to encode board positions (listing 2.4) where black, white and every piece type fields are bitboards, the move field is the moves played count, the rule50 is the fifty-move rule count and the turn field indicates which player must make the move in the current position.
struct {
    uint64 t white, black;
    uint64 t kings, queens, rooks, bishops, knights, pawns;
    uint16 t move;
    uint8 t castling, rule50, ep;
    bool turn;
} pos;

Listing 2.4: Data structure to encode board on fathom

Bitboard is a board representation method that uses a set of 64 bits where each bit identifies the occupancy of each square for one piece type or colour.

Since Tablebases can evaluate a 4-piece game position more accurate than a simulation. Syzygy 4-men Tablebases will be used to improve the precision of MCTS simulations and to reduce the length of the simulations. They will also be used to evaluate expanded nodes with 4-pieces game positions, during the expansion stage of MCTS. For accessing Syzygy Tablebases, Fathom will be adapted and used in the following work.
Chapter 3

State of the Art

In this chapter, the state of the art on MCTS is reviewed. MCTS can be optimized by adding application specific strategies and/or by executing the MCTS in parallel.

The first section will present and analyse existent works on the use of MCTS for specific applications. The primary focus, in this section, is to compare and verify the performance achieved on the strategies proposed.

The second and third section aims to review proposed solutions on the parallelization of MCTS. The second section focus on software parallelizations and the third section reviews Heterogeneous Hardware implementations.

3.1 MCTS Applications

Almost every published articles about MCTS is applied in games, so the works reviewed in this section are mostly game-related, although two non-game applications are also analysed. There is going to be an analysis on published implementations of MCTS and a small description of each application will also be given, in the following applications:

- MS Pac-Man;
- Go;
- Lines of Action;
- Chess;
- Einstein Würfelt Nicht (EWN);
- Detecting wolves in a matching algorithm;
- Physical Travelling Salesman problem.

MS Pac-Man is a popular arcade video game where the player should avoid the ghosts and earn points by eating pellets. The game ends when player “is eaten” by one of the ghosts.
Go is a two-player game with perfect information, played in a 19x19 board (there are variants that use smaller boards) with black and white stones, a different colour for each player. The objective is to surround the opponent's stones and when it is successfully done, those surrounded stones are “captured”. The game ends when a player gets to a position where is not possible to make any more points. Every stone present in the board represents a point.

Lines of Action is a two player game with perfect information, played on a 8x8 board where each player has 12 pieces to play and there is a specific way to move the pieces on board. The objective is to connect all the 12 pieces into a single group, these connections can be horizontal, vertical or diagonal. The game ends when a player gets reduced to one only piece, when a player connects all pieces or when is not possible for a player to make a move.

EWN is a 5x5 board game where each player controls six numbered pieces, placed in a triangular pattern, when game starts. Each piece can move either horizontally, vertically or diagonally. On each turn, the player rolls a dice to decide which piece is going to be moved. If moved to a position which already contains an opponent's piece this one is captured. The game ends when a player occupies the corner field of his/her opponent or when a player captures all the opponent's pieces.

A wolves attack is an attempt to impersonate a victim in an image-based biometrics authentication system, by feeding input data that can be falsely accepted as a match.

Physical Travelling Salesman Problem (PTSP) is an extension of the Travelling Salesman Problem, a well known optimisation problem. In PTSP, a salesperson starts in the centre of a map and has to visit \( n \) cities only once, using the shortest route possible, returning to the starting point in the end. These cities are usually distributed randomly within some rectangular area. The solution may be obtained using a 1-player MCTS, such that, each time the tree depth increases, a new city is chosen to be visited. A win is achieved, when all the cites are visited using the shortest path.

### 3.1.1 Applications Strategies Analysis

Ms Pac-Man [20], a Chess AI player [21] and MANGO (an AI Go player) [22] in 2014, 2012 and 2007, respectively, have implemented successfully their games using MCTS with UCT as a tree-searching algorithm.

The primary focus in MANGO was to verify how much their algorithm had improved by applying some techniques (as the use of heuristic functions) and not to achieve performance using UCT in MCTS.

Pac-Man is a game where fast decisions are required, as MCTS depends on the number of simulations to achieve a reliable outcome, the lack of computing time harms the correctness of MCTS. But with some strategies on simulation, they’re able to improve MCTS outcome, achieving a suitable and effective algorithm for Ms Pac-Man.

In the AI Chess player [21] the primary target was to test MCTS performance when applied to Chess and examine how modifications to the base algorithm can improve it, even with past indicators that MCTS is not suitable for the game of Chess.

MCTS was also applied to generate several wolves candidates in [23], starting with an image with
all pixels undecided, fixing one pixel each time the tree depth increases (see figure 3.1). In this case, when simulations are performed, random values are chosen to the undecided pixels. Every time an image pattern returns a false positive in the matching algorithm, which is a image pattern accepted by the matching algorithm which shouldn’t be, is a MCTS winning state. In this approach, authors from [23] were successful and suggested that their implemented method would be useful for developers of matching algorithms.

![Tree structure used in [23] to find wolves candidates (from [23]).](image)

Figure 3.1: Tree structure used in [23] to find wolves candidates (from [23]).

When heuristic functions were applied on node selection, MC-LOA [24], a 2010 Lines of Action playing AI, MANGO, an AI Chess program [21] and a PTSP solver [25] showed good results (first 3 shown in table 3.1). Every application used Progressive Bias (equation 2.2) in MCTS. MANGO also used Time-Expensive Heuristics to avoid speed reduction and Progressive Unpruning, which is a technique that initially reduces artificially the branching factor of the MCTS tree and increases it progressively. These Time-Expensive Heuristics worked in such a way, that when a certain threshold of games has been played through a node, the algorithm would stop computing the heuristics. The Chess AI program also used Heavy Playouts, Decisive Moves and endgame Tablebases to take advantage of domain knowledge to create more accurate simulations. The Heavy Playouts employ a trade-off between the use of heuristic functions and random plays to choose a move during simulations. Decisive Moves technique performs a test every time a move leads to a situation that when the opponent’s king is in a captured position, this move would only be performed if the opponent’s king was not able to escape (checkmate).

<table>
<thead>
<tr>
<th>Application</th>
<th>MCTS with UCT</th>
<th>MCTS using Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANGO</td>
<td>25%</td>
<td>58%</td>
</tr>
<tr>
<td>AI Chess player</td>
<td>17%</td>
<td>99%</td>
</tr>
<tr>
<td>MC-LOA</td>
<td>Non existent</td>
<td>46%</td>
</tr>
</tbody>
</table>

Table 3.1: Winning rate attained on MCTS based AI players.

As benchmark, MANGO played 200 games against GNU Go (3.7.10) at level 10. In GNU Go, a
strong Go program, it is possible to adjust the AI strength, by changing its level.

In MC-LOA was used MIA, as benchmark, which is a world-class Lines of Action program. MIA won the eighth (2003), ninth (2004) and eleventh (2006) Computer Olympiad.

In the AI Chess player [21], round-robin tournaments of blitz chess with five minutes clock time per side was used, where the players were several versions of itself. Table 3.1 presents the weakest version (MCTS with UCT) and the strongest (MCTS using heuristics in selection and simulation stages). Even with the increase on the performance, it is referred in [21], that their strongest version still does not stand a chance against Minimax (algorithm used on major Chess programs) based competitors.

A PTSP solver implemented in [25] also showed improvements when heuristics were applied on selection stage. In their solver, they composed a "MCTS UCT" implementation, which was a standard UCT, a "Centroid Heuristic MCTS" implementation, when it uses only the heuristic function to perform the selection stage and a "Centroid MCTS & UCT" implementation, when a balance is created between the heuristic and UCT to perform the selection stage. They concluded that is possible to achieve the best results when a balance between UCT and the heuristics is performed.

Considering these implementations, it is possible to conclude that adding heuristic functions to MCTS selection becomes a huge asset.

Taking into account the PTSP solver, it is also possible to figure the importance to create a balance between the standard UCT and heuristic functions, since that "Centroid MCTS & UCT" showed much better results than "Centroid Heuristic MCTS" or "MCTS UCT".

Nevertheless it must always be regarded the time consumption as adding heuristics turns MCTS slower. The usage of time-expensive heuristics, as in MANGO [22], it is a good technique to reduce the computation time of MCTS.

### 3.1.2 AlphaGo

AlphaGo was the first Go program to win against a Go champion on a 19x19 board, achievement that was not been accomplished by any other Go Program, until now, due to the size of the search space. This program combined Monte Carlo tree search and deep neural networks with a huge computational power.

This program uses "Value networks" to evaluate board positions and "Policy networks" to select moves. The "Value Network" evaluates the board positions in the tree in order to guide the MCTS selection. The "Policy networks" allows to perform a quick action with domain-specific knowledge on game states, while simulating, and on the expansion stage of MCTS where it provides an superior action to perform on a certain game state.

Its strongest version used 40 search treads on MCTS, 1202 Central Processing Units (CPUs) and 176 Graphics Processing Units (GPUs) distributed on several machines. To avoid different search threads selecting the same nodes it was used the "Virtual Loss" technique, detailed on section 3.2.3.
3.2 Parallelization

Every simulation in MCTS is independent from the others, therefore executing the simulations in parallel is relatively simple to implement efficiently and does not require any specific synchronization mechanisms. The parallelization can also be extended to the tree search phases by sharing or splitting the tree. Figure 3.2 summarizes the main parallelization methods for MCTS. Leaf parallelization refers to methods where only the simulation phase is parallelized, Root parallelization refers to methods where distinct trees are developed in parallel and Tree parallelization designates methods that execute the MCTS stages in parallel on a unique shared tree. This section introduces previous works on such parallelization methods, presenting and evaluating their results.

3.2.1 Leaf Parallelization

Leaf parallelisation is the simplest method to parallelize MCTS, since it does not require any type of memory synchronization. In this method there is one thread for selection, expansion and backpropagation phases, and all other available threads are used to simulate. Once a node is selected to be simulated on MCTS, several simulations are executed of that state node, each one in a different thread. This way is possible to achieve a better accuracy on each simulated node.

A disadvantage of using this method is that since each simulation has a distinct execution time, the algorithm must wait until the longest simulation finishes. There is not shared information between threads, so for instance, if 10 parallel simulations are running and 9 finish faster with a loss, the algorithm must wait for the longer one to finish, which probably will end in a loss as well.

Kato and Takeuchi [27] have evaluated this method implementing it in a Go playing program (FUDO GO) with a client-server network architecture. With this method they were able to increase from 35% (un-parallelized version) to 67% the winning percentage. These values were attained using a limit of 0.8 seconds to make a move (time to run all the four stages of MCTS).
3.2.2 Root Parallelization

In Root parallelization, the algorithm expands one tree in each thread, which means that the root is parallelized, and one tree is developed from that root in each thread. Once the tree search is stopped all the children of the root are merged, thus all these children statistics (in the main thread) are updated with the information from other clone nodes on other trees (in the remaining threads). Ultimately, the best move is chosen using the statistics from those merged root children.

In MC-LOA \[24\], using this method, against MIA they were able to raise their winning percentage from 45% to 56%, using two threads, and to 60%, using four threads, showing better results while compared to their single-threaded version.

3.2.3 Tree Parallelization

In Tree parallelization, the algorithm runs multiple simulations on a unique shared tree. Sharing the tree requires the inclusion of specific access synchronization mechanisms. \[26\] proposed the use of mutexes, globally or locally, to synchronize the parallel accesses to the tree information.

The use of mutexes gives exclusive access to a tree resource, preventing other threads from simultaneously accessing the same memory locations. When using a global mutex (mutual exclusion), only one thread can traverse the tree (selection, expansion, and backpropagation phases) at a time, but the others may concurrently perform simulations from different leaf nodes.

Tree parallelization with local Mutual Exclusions ( Mutexs) allows several threads to access the tree simultaneously, but every time a thread visits one tree node that node is locked using a local Mutex. Since many visits are performed during MCTS, it is beneficial to use fast-access mutexes to achieve the maximum performance.

Another problem in this parallelisation method, beside data synchronization, is the possibility that a thread visits a node for which a simulation is being performed. In this event, to perform selection the algorithm has to wait until the the simulation is over to make the selection, taking into account that simulation outcome. In an attempt to solve this issue, \[26\] presented “Virtual Loss” which consisted of: before the simulation is finished other threads see the statistics of that node, as if that simulation ended in a loss. With this technique, the algorithm can continue even if it reaches a node that is performing a simulation, turning the algorithm quicker. This resulted in a successful technique.

In an implementation of Tree parallelization with a global Mutex on EWN \[28\], the execution was evaluated with 18 and 48 threads, and with different limit times to make a move (results shown in figure 3.3). The results were obtained playing against a un-parallelized version of itself.

It is possible to verify that a higher number of threads always outperform a smaller one and that, as the thinking time raises the win rate decreases. In this method, the selection stage is always performed by a single thread so, threads quantity does not affect the depth of search tree. A lower time limit shorts the simulations, therefore increasing the depth of the search tree, this can be one reason for the decrease in the win rate when thinking time raises.
3.2.4 Comparison Between Parallelization Methods

This section will compare the parallelization methods, based upon results from [26], which compares the winning percentage of the several parallelization methods (table 3.3), and [23], that compares the speed of search instead (table 3.2). In both works MCTS was tested using UCT as tree-selecting algorithm.

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf Parallelization</td>
<td>1226</td>
<td>676</td>
<td>615</td>
<td>618</td>
<td>630</td>
<td>619</td>
<td>626</td>
<td>634</td>
</tr>
<tr>
<td>Root Parallelization</td>
<td>505</td>
<td>253</td>
<td>191</td>
<td>154</td>
<td>135</td>
<td>113</td>
<td>104</td>
<td>91</td>
</tr>
<tr>
<td>Tree Parallelization with local mutexes</td>
<td>514</td>
<td>299</td>
<td>207</td>
<td>163</td>
<td>133</td>
<td>109</td>
<td>105</td>
<td>98</td>
</tr>
<tr>
<td>Tree Parallelization local mutexes with local mutexes and Virtual Loss</td>
<td>511</td>
<td>297</td>
<td>198</td>
<td>152</td>
<td>131</td>
<td>110</td>
<td>105</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 3.2: The speed of decision-making on the parallelization methods (in ms) [23].

In both works, the Tree parallelization method shows slightly worse results than Root parallelization, which is an indication that the Mutex synchronization overhead penalizes the shared-tree parallelization. Also, Leaf parallelization always shows significantly worse results, which is due to the fact that the implementations wait for all parallel simulations to end before continuing to the next MCTS stage.

Despite a few inconsistencies on the number of games played on some of the experiments, presented in table 3.3 (which justifies the significant variance on the confidence interval values within the same method), can be concluded that Root parallelization outperforms any other method, both in accuracy (table 3.3) and execution speed (table 3.2). The authors in [26] suggest that Root parallelization strength comes from being a more efficient method of MCTS parallelization but they also indicate that this method prevents MCTS from remaining too long on a local optima (problem existing in UCT). They consider as well that, if the tree-searching algorithm were able to perform a better balance between the exploitation and exploration of the MCTS tree, then the Tree parallelization would be, at least, as good as the Root parallelization.
<table>
<thead>
<tr>
<th>Number of threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Leaf Parallelization</strong></td>
<td>Winning Percentage</td>
<td>26.7%</td>
<td>26.8%</td>
<td>32.0%</td>
</tr>
<tr>
<td></td>
<td>Confidence interval</td>
<td>2.2%</td>
<td>2.0%</td>
<td>2.8%</td>
</tr>
<tr>
<td></td>
<td>Number of Games</td>
<td>2000</td>
<td>2000</td>
<td>1000</td>
</tr>
<tr>
<td><strong>Root Parallelization</strong></td>
<td>Winning Percentage</td>
<td>26.7%</td>
<td>38.0%</td>
<td>46.8%</td>
</tr>
<tr>
<td></td>
<td>Confidence interval</td>
<td>2.2%</td>
<td>2.2%</td>
<td>2.2%</td>
</tr>
<tr>
<td></td>
<td>Number of Games</td>
<td>2000</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td><strong>Tree Parallelization with global mutex</strong></td>
<td>Winning Percentage</td>
<td>26.7%</td>
<td>31.3%</td>
<td>37.9%</td>
</tr>
<tr>
<td></td>
<td>Confidence interval</td>
<td>2.2%</td>
<td>2.2%</td>
<td>2.2%</td>
</tr>
<tr>
<td></td>
<td>Number of Games</td>
<td>2000</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td><strong>Tree Parallelization with local mutex</strong></td>
<td>Winning Percentage</td>
<td>26.7%</td>
<td>32.9%</td>
<td>38.4%</td>
</tr>
<tr>
<td></td>
<td>Confidence interval</td>
<td>2.2%</td>
<td>2.2%</td>
<td>2.2%</td>
</tr>
<tr>
<td></td>
<td>Number of Games</td>
<td>2000</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td><strong>Tree Parallelization with local mutex and VL</strong></td>
<td>Winning Percentage</td>
<td>26.7%</td>
<td>33.8%</td>
<td>40.2%</td>
</tr>
<tr>
<td></td>
<td>Confidence interval</td>
<td>2.2%</td>
<td>2.2%</td>
<td>2.2%</td>
</tr>
<tr>
<td></td>
<td>Number of Games</td>
<td>2000</td>
<td>2000</td>
<td>2000</td>
</tr>
</tbody>
</table>

Table 3.3: Winning percentage and confidence interval between parallelization methods [26].

### 3.3 Heterogeneous Hardware Implementations

This section focuses mainly on parallelization methods used in Heterogeneous Hardware Implementations. It analyses the systems on where the applications were implemented and if any special care is required when parallelizing MCTS, having in consideration the systems characteristics.

#### 3.3.1 GPU and Many-Core

This subsection focuses in one implementation on the TSUBAME 2.0 (equipped with a NVIDIA TESLA C2050 GPUs and Intel Xeon X5560 CPUs) [29] and other on the Intel Xeon Phi [30]. Both systems offer a massive parallelization power and therefore scheduling the threads becomes a major issue. So, this subsection will also analyse how the authors proposed to perform this scheduling.

The work in [30] shows the use of the Tree parallelization method with local mutexes. The Xeon Phi processor contains 61 cores and each core supports four Vector Processing Units (VPUs), this summarizing a total of 244 (=61x4) possible threads. The Tag Directorys (TDs) are used to look up cache data distributed among the cores. There are a total of 8 distributed memory controllers (MC). The connection between all units is performed through a bidirectional ring interconnect (figure 3.4).

In this implementation the maximum number of iterations (where one iteration is an execution of all four stages of MCTS) was split by tasks, this way every task is responsible for running a certain number of MCTS iterations. Every task was then processed by a hardware thread. They tested their implementation with a different number of tasks. If there are more tasks than threads, once a hardware thread ends a task, it will start another, until the program completes all tasks required. In this method, every new task is added to a queue.
They evaluated different methods to manage the threads scheduling on the Intel Xeon Phi and concluded that a simple First In First Out (FIFO) queue provided the best results. In this method, every time a thread becomes available, it will start a task from the queue, using the principle of first-come first-served.

This implementation was tested with a total of 1,048,576 playouts (maximum number of total iterations). As a result when 4096 tasks were used, they reached the maximum speedup of 47 times (each task performed 256 iterations). This speedup was attained by comparing it to the speed of a version on where only one task was used, therefore without any parallelism.

The work in [29] implemented an approach using the Leaf-Root parallelization methods working simultaneously, where the Leaf parallelization is implemented in the GPU and the Root parallelization in the CPU. Their implementation used a one CPU and one GPU from the TSUBAME 2.0. For Root parallelization, they grouped a fixed number of simulations per tree, naming it block size. In their implementation, the authors obtained the best results with 112 parallel trees and a total of 14336 GPU threads, which corresponds to 128 block size (simulations per tree). When fewer GPU threads were used, the authors obtained better results using smaller block sizes (32 simulations per tree), meaning more parallel trees and less simulations per tree (figure 3.5).
The higher number of available threads does not seem to bring advantage to Leaf parallelization method when compared to an approach of Leaf+Root parallelization methods working simultaneously, once it never outperforms the winning rate in [29]. In [30], it is also possible to realize that with the high number of threads comes a need to perform a careful thread scheduling to obtain a better outcome.

### 3.3.2 FPGAs

This subsection presents two FPGA architectures, one implemented (figure 3.6) and one proposed (figure 3.8), both game related.

![Figure 3.6: Implemented architecture in [31] (from [31]).](image)

The MCTS Blokus Duo implementation in [31], uses a Root+Leaf parallelization methods (however with the name of “Slow-Tree-Parallelization”) and the architecture (figure 3.6) consists of five main parts:

- Communication module, which is responsible for the communication with the host program;
- Global Map, which contains the current game state. While the best move is not chosen and performed, this state corresponds to the initial game state;
- WatchdogTime, which oversees the time remaining to choose a best move;
- Global Best Move, which contains the best move learnt from MCTS Solver modules at each state. The best move learnt is acquired using the best move of each MCTS Solver, weighting each one of them, to collect the global best move;
- MCTS Solver modules, which will receive the game state from Global Map and perform MCTS on that state. This is a hardware implementation of root parallelization, since several new trees are created and perform MCTS on each independent Solver module, therefore parallelizing on the Global Map current state.

The MCTS solver module (figure 3.7) stores all the tree nodes in the Main Memory, which is implemented using true dual-port Random Access Memory (RAM) with 32-bit words. The MCTS Controller oversees the selection, expansion, and backpropagation MCTS stages accessing the information stored in the memory. MCTS Engines perform the simulation stage, receiving the state from the MCTS Controller and returning the outcome to it.
The MCTS Engines were implemented with four main components:

- A Local Map, which stores the initial game state and updates it through the simulations;
- A Find Move, which with a given game state, finds possible moves and sends them to Simulation Controller;
- A Simulation Controller, which randomly chooses a move to perform, sends the new game state to Score Calculation and sends to Local Map the new game state (updating the current game state);
- A Score Calculation, which evaluates a game state.

[32] proposes a hardware/software architecture, in which only the simulation stage is performed in hardware. This proposed architecture (figure 3.8) is dedicated to Trax game, where pattern recognition in an image is required. The pattern contains two types of paths (identified by its colour) making it possible to have several paths of the same type on the same game state. This architecture consists in nine main parts:

- Universal Asynchronous Receiver-Transmitter (UART), which receives a data package containing a move in a game state. UART is used for asynchronous serial communication;
- Move Translator, which decodes the data package from UART to a format that the Main Processor is able to read;
- Path Memory, which stores all the existing paths in a game state, making it possible to simulate one path at a time;
- Main Processor, which, after receiving the move, this component will scan possible paths from the Path Memory and send them to the Board Simulator;
- Board Simulador, which applies actions on the received path and send them to the Solution Analyzer;
- GR (General Rater), which provides a summary of the potential of that path to win or lose, based on data store in it;
- PR (Pattern Recognizer), which performs a pattern recognition of the input data;
- Score Synthesizer, which synthesises the information received from the GR and all the Pattern Recognizers. This component only stores a new score if the new value is superior to the one currently stored. This way, even though the BS and the PRs run all the possible actions on the paths, only the best score is stored;

- Solution Register, which stores the best action on the Path, when the Score Synthesizer finds a new best value;

- SA (Solution Analyser), which is the hardware block responsible for analysing a game state and to evaluate it.

Figure 3.8: Proposed architecture in [32] (from [32]).

In this proposed architecture, when the UART receives a state, the move translator decodes it and sends it to the MP. Then the BS gets ready for scanning possible solutions, according to all the paths stored in the Path Memory. When the BS computes a possible board state and finds a feasible solution, this one is sent to the SA to be evaluated. The SA analyzes the board state and computes a score for it. After that, the next path will be processed and the procedure is repeated. When all the possible solutions are scanned or when it runs out of time, the analysing process ceases and the current best solution is translated and sent to the host computer through the UART.

This proposed architecture exclusively implements the MCTS simulation stage and therefore the hardware architecture can be used as a co-processor unit in any kind of parallelization method. Using such multiple units it is possible to send them a state to be simulated by several units (leaf parallelization), or to manage multiple trees or a shared tree on the host computer and send it the states of different nodes (tree or root parallelization).

These works show that it is possible to implement MCTS on a FPGA.

A special care is required when the tree is stored in hardware therefore, an architecture, as the one on figure 3.8, in which only the simulation is performed, might be a simpler way of implementing MCTS using a hardware-software architecture.
Chapter 4

MCTS Software Implementation

The software development was divided in two main steps. Firstly, the MCTS functions were developed and applied to the Tic-tac-toe game. Secondly, the program functionality was extended, to be applied to Chess endgames, by adding heuristic-based selection and developing the required more complex move generator and evaluation function. The Chess endgames MCTS implementation was limited to a maximum of 1 Rook and 4 Pawns for each player. This division allowed to perform first a simpler version of the MCTS algorithm, since the Tic-tac-toe require a simple move generation function, in order to focus on developing the functions to perform MCTS. The Tic-tac-toe version was implemented totally heuristics free, such that every move selected during simulations was fully random and the tree nodes were selected using UCT (formulation 2.1). Then, the moves generator was replaced by the more complex one as required to implement the game of Chess and a board evaluator function added to include Chess-specific heuristics in the simulation, expansion and selection stages of MCTS, as explained below (subsection 4.2). Endgame Tablebases were also to the MCTS Chess implementation.

4.1 Tic-tac-toe

Every tree node used on the Tic-tac-toe implementation stores the information of the board, the number of wins, the number of plays, side-to-play, pointers to the children nodes and one pointer to the parent node. The node data structure is shown in listing 4.1.

```c
struct {
    int Board[9], Plays, Side-To-Play;
    float Wins;
    struct node *Child[9], *Parent;
} node;
```

Listing 4.1: Data structure for the tree nodes in the Tic-tac-toe MCTS implementation

The algorithm 2 shows how the selection stage of MCTS is performed. First, the root is selected. Then, the children with the biggest UCT value (most promising children) are successively selected, until a leaf node is reached. Once the selection ends, the leaf node is expanded.
Algorithm 2 Tic-Tac-Toe MCTS Selection Stage

1: procedure SELECTION(Root)
2: 
3: Iterations = 0
4: 
5: while Iterations < MaxIterations do
6: 
7: Iterations ++
8: 
9: Node = Root
10: 
11: while Node ≠ Leaf do
12: 
13: for Every Node.Child do
14: 
15: if UCT(Node.Child) > UCT(Node) then
16: 
17: Node = Node.Child
18: 
19: Expansion(Node)
20: 
21: return

The algorithm 3 illustrates how MCTS expansion is performed. The CheckGameState verifies if the game reached an ending. This function returns an integer value that depends if the game ended or not. In the case of a node with an ended game is selected, the state of that node is directly backpropagated. For this, the state is converted in an integer value by the GameResult function. The value returned by this function depends if the game ended in a win (1), draw (0) or loss (-1) for the root player. In the case of a node with a game that is not over yet, the function PlayoutGenerator returns all possible moves. And the AddNewNode function adds the children nodes, one for every possible playout (all children nodes are immediately created once their parent is selected to be expanded). Every time a child node is added, its side-to-move field is updated in order to change the player’s turn. Once all children nodes are created, the board position is simulated the user-defined number of times. Finally, the ending state is backpropagated.

Algorithm 3 Tic-Tac-Toe MCTS Expansion Stage

1: procedure EXPANSION(Node)
2: 
3: Result = 0
4: 
5: State = CheckGameState(Node.Board)
6: 
7: if State ≠ EndingState then
8: 
9: Playouts = PlayoutGenerator(Node.Board, Node.Side-To-Move)
10: 
11: for i = 0; i < Playouts; i = i + 1 do
12: 
13: AddNewNode(Node.Child[i], Playouts[i], i)
14: 
15: for j = 0; j < Simulations; j + + do
16: 
17: Result += Simulation(Node.Board, Node.Side-To-Move)
18: 
19: Result = GameResult(State)
20: 
21: Backpropagation(Node, Result)
22: 
23: return

The MCTS simulation (shown in algorithm 4) receives as arguments the board position to be sim-
ulated and the side-to-move. All possible moves are generated and a random one is chosen and executed. Once the move is completed, the player’s turn is updated. This process is repeated until the Check_GameState verifies that the game reached an ending situation. Once ended, the state is converted by the GameResult function.

Algorithm 4 Tic-Tac-Toe MCTS Simulation Stage
1: procedure SIMULATION(Node.Board, Node.Side-To-Move)
2:   Board = Node.Board
3:   Player = Node.Side-To-Move
4:   while State ≠ EndingState do
5:     Playouts = Playout_Generator(Board, Player)
6:     Board = Apply_Random_Move(Playouts)
7:     Player = Change_Turn(Player)
8:     State = Check_GameState(Board)
9:   end while
10:  Result = GameResult(State)
11: return Result

The MCTS backpropagation (shown in algorithm 5) updates all the parent’s nodes, starting from the node just simulated until it reaches the root. The number of plays and wins for every node are updated.

Algorithm 5 Tic-Tac-Toe MCTS Backpropagation Stage
1: procedure BACKPROPAGATION(Node, Result)
2:   Node = Node
3:   while Node ≠ NULL do
4:     Node.Wins += Result
5:     Node.Plays += Simulations_No_Per_Expansion
6:     Node = Node.Parent
7: return

While the iterations do not exceed the user-defined value, these four MCTS stages are repeatedly executed.

After the implementation was completed, several tests were performed. The Max Child criteria was used to select the best action in all tests, which means that once the number of maximum iterations is reached the root child with the highest ratio of wins per plays is chosen. First, the best action chosen by MCTS was checked for three specific game states, in order to verify if the algorithm was able to choose the right move. Each game state was chosen to check the algorithm behaviour in distinct situations.

Initial Position 1 (figure 4.1) was used to verify if the algorithm chooses the central square, since this is the decision that leads to a greater probability of victory. Position 2 (figure 4.2) chosen to verify if the algorithm places the missing mark in order to obtain a victory. And position 3 (figure 4.3) chosen to verify if the algorithm blocks the opponent’s possible victory.
Then, a random move player algorithm was created to compete against MCTS. These games were performed in order to verify how the number of MCTS iterations and simulations per expansion affects the algorithm. As shown in table 4.1, the MCTS strength increases with the iterations and simulations per expansion increase. A larger number of iterations implies a bigger search tree and simulations quantity, therefore improving, as expected, MCTS. Also, since MCTS uses the simulations results to select the most promising nodes, a larger number of simulations leads to a more precise selection, and consequently to a better accuracy.

<table>
<thead>
<tr>
<th>MCTS iterations</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>4219</td>
<td>2494</td>
<td>1958</td>
<td>1566</td>
<td>1459</td>
</tr>
<tr>
<td>100</td>
<td>2547</td>
<td>1475</td>
<td>1212</td>
<td>1093</td>
<td>979</td>
</tr>
<tr>
<td>200</td>
<td>883</td>
<td>542</td>
<td>353</td>
<td>350</td>
<td>276</td>
</tr>
<tr>
<td>500</td>
<td>127</td>
<td>13</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1500</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: MCTS total losses in 100 000 games played against a random algorithm.
To perform a performance analysis, the runtime averages of all MCTS stages were measure for each iterations quantity and different simulations per expansion, as shown in table 4.2. Since this game has a simple moves generation function and short simulations, the UCT calculations is the most time-consuming process in the algorithm. Therefore, the selection is the longest stage, as can be verified by analysing table 4.2. In this table, the selection, expansion and backpropagation stages timings are omitted, for more than one simulation per expansion, since they are the same.

<table>
<thead>
<tr>
<th>MCTS Iterations</th>
<th>Simulations per expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Selection</td>
<td>Expansion</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>500</td>
<td>36</td>
</tr>
<tr>
<td>1500</td>
<td>130</td>
</tr>
</tbody>
</table>

Table 4.2: MCTS stages execution time depending on iterations and simulations per expansion (in ms).

For this Tic-tac-toe implementation, a MCTS execution with less iterations and more simulations per expansion can optimize the execution time maintaining the strength of MCTS. The a priori expansion of all nodes will not be implemented in the following work, since about 60% of the search tree nodes are never used. Also regarding the memory use, for applications with a higher number of possible moves and/or a larger board, the use of pointers for all children nodes and the board array representation would necessarily lead to a high memory usage.

### 4.2 Chess Endgames

Firstly, the nodes data structure was optimized and adapted to be applied to Chess endgames. For this, the move performed on the parent node game position replaced the array board representation. Also, a first child and a next brother pointers replaced the multiple children pointers. This modification changed the search tree structure, as shown in figure 4.4.

![Figure 4.4: Old (left) and new (right) tree structure.](image)

A move count field was added to the data structure to check the fifty-move rule. The fully expanded node field was also added to allow the creation of one node at a time. The node data structure used is represented in listing 4.2. The described optimizations reduced the data structure size from 752 bytes (with a maximum of 40 children) to 28 bytes.
struct {
    uint8_t From_Square, Target_Square;
    uint8_t Side_To_Move, Fully_expanded, Fiftycounter;
    unsigned int Plays, Wins;
    struct node *Child, *Brother, *Parent;
} node;

Listing 4.2: Node data structure

Secondly, to apply the MCTS algorithm to Chess endgames, new moves generator and board evaluator functions, both Chess-specific, were developed. As referred in subsection 2.3.1, these functions were adapted from Faile [13].

The move generator does not consider castling neither en passant pawn moves, since in endgame situations there are not usual. Also, the Pawn is only promoted to Queen, since this is the most valuable piece and therefore the most common promotion. The algorithms 6 and 7 show how the moves are generated for a white sliding (Rook) and a white non-sliding piece (King). For the other white pieces and black pieces the methodology is the same, changing only the offsets applied and the rank where the pawn is promoted.

Algorithm 6 White Pawn and Ehite Rook Moves Generator

```
1: procedure MOVE GENERATOR(move_s moves[], int * num_moves, int board[], int side-to-move)
2:     if side-to-move = whites turn then
3:         for every white piece on board do
4:             piece = new piece
5:             from = piece position
6:             if piece = white pawn then
7:                 if rank(from) = 2 & board[from + 24] = nopiece & board[from + 12] = nopiece then
8:                     push_non_slide(&moves[0], num_moves, from, from + 24, 0, board)
9:                 if board[from + 13] = Black_piece then
10:                    push_non_slide(&moves[0], num_moves, from, from + 13, 0, board)
11:                 if board[from + 11] = Black_piece then
12:                    push_non_slide(&moves[0], num_moves, from, from + 11, 0, board)
13:                 if board[from + 12] = nopiece then
14:                    push_non_slide(&moves[0], num_moves, from, from + 12, 0, board)
15:             if piece = white rook then
16:                 push_slide(&moves[0], num_moves, from, from + 12, board)
17:                 push_slide(&moves[0], num_moves, from, from - 12, board)
18:                 push_slide(&moves[0], num_moves, from, from + 1, board)
19:                 push_slide(&moves[0], num_moves, from, from - 1, board)
20:         return
```
Algorithm 7 White Pieces Moving Functions

1: procedure PUSH_SLIDE(move_s moves[, int * num_moves, int from, int target, int board[ ]])
2:     offset = target − from
3:     hit_piece = FALSE
4:     while !hit_piece & board[target] ! = boarder do
5:         if board[target] = npiece then
6:             Add_new_move(&moves[0], num_moves, from, target, npiece, 0)
7:             target += offset
8:         else if board[target] = black piece then
9:             Add_new_move(&moves[0], num_moves, from, target, board[target], 0)
10:            hit_piece = TRUE
11:        else
12:            hit_piece = TRUE
13:     return

14: procedure PUSH_NON_SLIDE(move_s moves[, int * num_moves, int from, int target, int board[ ]])
15:     if board[from] = white pawn then
16:         if board[from] = white pawn & rank(from) = 7 then
17:             Add_new_move(&moves[0], num_moves, from, target, board[target], white_queen)
18:         else
19:             Add_new_move(&moves[0], num_moves, from, target, board[target], 0)
20:     else
21:         if board[target] = npiece then
22:             Add_new_move(&moves[0], num_moves, from, target, npiece, 0)
23:         else if board[target] = black piece then
24:             Add_new_move(&moves[0], num_moves, from, target, board[target], 0)
25:     return

26: procedure ADD_NEW_MOVE(move_s moves[, int * num_moves, int from, int target, int capture, int promoted])
27:     moves[+num_moves].from = from
28:     moves[+num_moves].target = target
29:     moves[+num_moves].captured = captured
30:     moves[+num_moves].promoted = promoted
31:     (+num_moves) + +
32:     return
The move generation algorithm does not verify if any of the generated moves leaves the side-to-move king in check. Thus, a function was developed to remove those from the initial set.

The evaluation function is used to add Chess-specific domain knowledge during the selection, expansion and simulation stages of MCTS. This function evaluates board positions according to algorithm 1 (subsection 2.3.1). However, a new condition was added to the evaluation (shown in algorithm 8), which applies a penalty whenever an unprotected piece is moved into a square that can be captured. In this occasion, the attacked piece value is subtracted from the final evaluation.

Algorithm 8: Penalty in Evaluation

1: procedure EVALUATION PENALTY(Moved_Piece_Position, Board, Side-To-Move, Score)
2:      Piece_Attacked = Check_Atk(Moved_Piece_Position, Side-To-Move)
3:      Piece.Protected = Check_Atk(Moved_Piece_Position, !Side-To-Move)
4:      if Piece.Attacked = TRUE & Piece.Protected = TRUE then
5:         Score = Score - Piece_Value
6:      return Score

A. Sadybakasov proposes in [33] a strategy to add domain knowledge during the simulation stage of MCTS. For this, the moves are evaluated according to a given probability \( p \). This means that the algorithm only computes the evaluation for \( p\% \) of the generated moves. In all other cases, the algorithm chooses a pseudo-random move. The algorithm 9 shows how is selected if the moves must be evaluated or not.

Algorithm 9: Original Random Model

1: procedure CHOOSE_MOVE(Num_moves)
2:      i = Pseudo_Random_Generator(from 0, to 100)
3:      if i < User Defined Probability then
4:         Chosen_Move = Best_Move
5:      else
6:         Chosen_Move = Pseudo_Random_Generator(from 0, to Num_moves)
7:      return Chosen_Move

A simplified model (shown in algorithm 10), targeting a possible hardware implementation, was developed to apply this strategy, based on the original model. In this model, two arrays loaded with pre-generated pseudo-random numbers are used. The Do_Eval array that contains 0 and 1 values in the same proportion that the moves must be evaluated. That means, if the user wants to choose the best-evaluated moves 30% of the times, then 30% of the values are 1 and the remaining 70% are 0. And the Random_Number array that contains pseudo-random values between 0 and 1. Both arrays are loaded before one or multiple MCTS execution(s). The moves are chosen by traversing both arrays while performing the presented arithmetic operations. Once the last element is reached, the first element from the array is selected again. This simpler approach, if implemented in hardware, will not require a random number generator. Since, two memories can be used to save the pseudo-random values.
Algorithm 10 Simplified Random Model

1: procedure CHOOSE_MOVE(Num_moves)
2:     Do_Eval[
3:     Random_Number[
4:     if Do_Eval[i] = 1 then
5:         Chosen_Move = Best_Move
6:     else
7:         Chosen_Move = Round(Num_moves * Random_Number[j])
8:     return Chosen_Move

This simplified random model was implemented in the simulation stage of MCTS. A probing function, adapted from Fathom [19], was implemented during the simulations, so that all 4-piece positions would be analysed by this function using the 4-men Syzygy tables. Before probing, the position is converted from the board array representation to bitboards. The algorithm 11 shows how the simulation stage is performed.

Algorithm 11 Chess Endgames MCTS Simulation Stage

2:     move_s Moves[
3:     Board = Input_Position
4:     Player = Node.Side-To-Move
5:     Fifty_Count = Node.Fiftycounter
6:     while Board ≠ EndingState do
7:         State = Check_GameState(Board, Fifty_Count)
8:     if State ≠ Ending State then
9:         Move_Generator(Moves, *Num_Moves, Board, Node.Player)
10:        Chosen_Move = CHOOSE_MOVE(Num_moves)
11:        if Chosen_Move = Best_Move then
12:           Chosen_Move = Evaluate_And_Choose_Highest_Move(Moves, Board)
13:        Do_move(*Board, *Capture, Moves[Chosen_Move].From, Moves[Chosen_Move].Target)
14:        Fifty_Count = Update_FiftyCount(Capture, Pawn_moved, Fifty_Count)
15:        Player = Update_Turn(Player)
16:     if State = Four_Pieces_On_Board then
17:        Bitboards = Array_To_Bitboards(Board)
18:        Table_Result = Probe_WDL_Tablebase(Bitboards, Player)
19:        Result = GameResult(Table_Result)
20:     else
21:        Result = GameResult(State)
22:     return Result
The function CheckGameState returns five possible values, which are: White side won; Black side won; Draw; 4-pieces on board; The game must continue. Once a final state is reached, the state or the result from the WDL tables are converted in a integer value by the GameResult function. This function returns an integer value that depends if the game ended in a win (1), draw (0) or loss (-1) for the root player.

In the selection stage, Progressive Bias was applied. One new tunable parameter ($W$) was added to control how much the heuristic function affects the node selection, as shown in the following equation:

$$ProgressiveBias = \frac{w_i}{n_i} + C \sqrt{\frac{\ln(n)}{n_i}} + W \frac{H_i}{n_i + 1}$$

(4.1)

Where the evaluation function is Chess-specific heuristic $H_i$ required by this technique. The algorithm 12 fully details how the selection stage is performed.

**Algorithm 12 Chess Endgames MCTS Selection Stage**

```plaintext
1: procedure SELECTION(Node Root, int StartingBoard[])
2:   Iterations = 0
3:   while Iterations < Max_Iterations do
4:     Iterations ++
5:     Board = StartingBoard
6:     Node = Root
7:     while Node Not Found To Expand do
8:       Child_Number = 0
9:       BestEval = -\infty
10:      if Node.Child = NULL then
11:         Expansion(Node, Board, 0)
12:      else
13:         Node = Node.Child
14:         while Node ≠ NULL do
15:           Do_move(*Board, *Capture, Node.From, Node.Target)
16:           if Progressive.Bias(Node, Board) > BestEval then
17:             BestNode = Node.Child
18:             BestEval = Progressive.Bias(Node, Board)
19:           Undo_move(*Board, *Capture, Node.From, Node.Target)
20:           Node = Node.Brother
21:           Child_Number ++
22:           if Node.Parent.Fully_expanded = 0 then
23:             Expansion(BestNode.Parent, Board, Child_Number)
24:           else
25:             Node = BestNode
26:           Do_move(*Board, *Capture, BestNode.From, BestNode.Target)
27:     end while
28:   end while
29: end procedure
```

40
Also during the selection stage, a board array representation is updated using the moves positions in each tree node. For this, to Do_Move and Undo_Move functions were created.

During the expansion stage, the Syzygy files are probed every time that a 4-piece game is expanded. The expansion algorithm expands one child at a time, as shown in algorithm 13. All possible moves are sorted in descending order, so that the best-rated moves are expanded first. Each added node contains the move performed in its parent node and the updated fifty count. Once the node children contains all legal moves added, that node is marked as a fully expanded node.

**Algorithm 13 Chess Endgames MCTS Expansion Stage**

1: procedure EXPANSION(Node, Board[], Child_Number)
2:   Num_Moves = 0
3:   Capture = 0
4:   Pawn_moved = 0
5:   Result = 0
6:   move_s_Moves[
7:   State = Check_GameState(Node.Board, Node.FiftyCount, Node.Side-To-Move)
8:   if State = Ending State then
9:     if State = Four_Pieces_On_Board then
10:        Bitboards = Array_To_Bitboards(Node.Board)
11:        Table_Result = Probe_WDL_Tablebase(Bitboards, Node.Side-To-Move)
12:        Result = GameResult(Table_Result)
13:   else
14:      Result = GameResult(State)
15:   Backpropagation(Node, Result)
16: else
17:   Move_Generator(Moves, *Num_Moves, Board, Node.Side-To-Move)
18:   Evaluate_And_Order_Moves(Moves, Board)
19:   From = Move[Child_Number].From_Square
20:   Target = Move[Child_Number].Target_Square
22:   Fifty_Count = Update_FiftyCount(Capture, Pawn_moved, Node.FiftyCounter)
23:   Added_Node = Add_New_Node(Node, From, Target, Child_Number, Fifty_Count)
24:   if Child_Number = Num_moves then
25:     Node.Fully_Expanded = 1
26:   for j = 0; j < Simulations_No_Per_Expansion; j + + do
27:     Result += = Simulation(Board, Added_Node.Side-To-Move, Added_Node.FiftyCount)
28:   Backpropagation(Added_Node, Result)
29: return
The backpropagation stage algorithm stayed the same as in the prior Tic-Tac-Toe implementation (subsection 4.1) where the number of plays and wins for every node are updated.

While the iterations do not exceed the user-defined value, the four MCTS stages are repeatedly executed.

4.2.1 Results

In order to verify if the simplified random model maintains the algorithm behaviour, the results of 10,000 simulations for two Chess positions were analysed. The tests were performed with 16K pseudo-random values in the Do_Eval array and 10K pseudo-random values of 12 bits (1 integer bit and 11 fractional bits) in the Random_Number array. Figures 4.5 and 4.6 show the white side winning percentage and black side winning percentage for the both models.

![Figure 4.5: Results of 10 000 simulations in Position 1 for the original and simplified random model.](image)

![Figure 4.6: Results of 10 000 simulations in Position 1 for the original and simplified random model.](image)

Different seeds were used in the pseudo-random generator for each of the tests performed. The simplified model results are congruent with the original model, thus validating this approach.

To verify the influence of the implemented techniques in the behaviour of the algorithm, 30 Chess
positions (shown in Appendix A) were tested. The tests were carried out for 4 versions of the MCTS, where different techniques were applied, presented below.

- **MCTS UCT**: The selection of nodes was performed using the UCT equation and the moves chosen during the simulation stage were completely random;

- **MCTS TB**: The selection of nodes was performed using the UCT equation and the Tablebases were used to evaluate all positions with 4 pieces, both in the expansion and simulation stages;

- **MCTS Eval**: The selection of nodes was performed using the Progressive Bias equation, the simplified random model was applied during the simulation stage and the best evaluated moves are expanded first;

- **MCTS Full**: The MCTS Eval version with the Tablebases applied.

The best moves chosen by MCTS were compared against the ones chosen by Stockfish, currently the world strongest Chess engine [34], in order to verify the number of moves that each version of the MCTS could find. This way, verifying if the implemented techniques positively affect the algorithm. All tests were performed with an exploration parameter of 1.41, heuristic weight in Progressive Bias of 0.0005, a 30% probability of choosing the best move during the simulations and using O2 compiling optimizations. These values were empirically chosen in order to maximize the precision of MCTS.

Table 4.3 presents the number of correct moves found by the different versions of the algorithm, for different numbers of iterations. The results were obtained using three simulations per expansion, since a lower number of simulations led to a low accuracy of results. The table also shows the results obtained using 10 simulations per expansion for the strongest version.

<table>
<thead>
<tr>
<th>MCTS Iterations</th>
<th>MCTS UCT</th>
<th>MCTS TB</th>
<th>MCTS Eval</th>
<th>MCTS Full</th>
<th>MCTS Full (10 Sim)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>1500</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>5000</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10000</td>
<td>8</td>
<td>8</td>
<td>13</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 4.3: Moves found by different versions of MCTS.

The use of the heuristic functions increased the number of best plays found, evidenced by the presented results of the MCTS Eval version. The use of Tablebases did not significantly influence the moves found by MCTS.

The execution time of each MCTS stage (average of the 30 positions) was also analysed for the positions tested, using one simulation per expansion, as shown in tables 4.4 and 4.5.
<table>
<thead>
<tr>
<th>MCTS Version</th>
<th>MCTS Stage</th>
<th>Selection [ms]</th>
<th>Expansion [ms]</th>
<th>Simulation [s]</th>
<th>Backpropagation [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCTS UCT</td>
<td></td>
<td>52</td>
<td>76</td>
<td>6.9</td>
<td>63</td>
</tr>
<tr>
<td>MCTS TB</td>
<td></td>
<td>54</td>
<td>76</td>
<td>5.2</td>
<td>66</td>
</tr>
<tr>
<td>MCTS Eval</td>
<td></td>
<td>79</td>
<td>78</td>
<td>6.2</td>
<td>63</td>
</tr>
<tr>
<td>MCTS Full</td>
<td></td>
<td>83</td>
<td>78</td>
<td>3.8</td>
<td>68</td>
</tr>
</tbody>
</table>

Table 4.4: MCTS execution time for (500 iterations).

<table>
<thead>
<tr>
<th>MCTS Version</th>
<th>MCTS Stage</th>
<th>Selection [s]</th>
<th>Expansion [s]</th>
<th>Simulation [s]</th>
<th>Backpropagation [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCTS UCT</td>
<td></td>
<td>2.1</td>
<td>1.5</td>
<td>138</td>
<td>1.5</td>
</tr>
<tr>
<td>MCTS TB</td>
<td></td>
<td>2.2</td>
<td>1.5</td>
<td>102</td>
<td>1.6</td>
</tr>
<tr>
<td>MCTS Eval</td>
<td></td>
<td>3.2</td>
<td>1.6</td>
<td>123</td>
<td>1.6</td>
</tr>
<tr>
<td>MCTS Full</td>
<td></td>
<td>3.6</td>
<td>1.6</td>
<td>75</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Table 4.5: MCTS execution time (10000 iterations).

The need of computing the heuristic function increased 36% the execution time of the selection stage and 6% the expansion stage. However, it reduced the execution time of simulation stage by 7%, since less moves are now performed during simulations (the random effect diminished). The Tablebases also reduced the execution time of the simulation stage by 26%. The MCTS Full is 43% faster than the MCTS UCT.

As expected, in the strongest version, the simulation stage consumes more than 93% of the total execution time of the algorithm. It was also verified that the high execution time of the simulation stage was due to the games and not due to the probing of Tablebases (less than 1% of the simulation stage execution time). So, the MCTS Full version using a dedicated hardware architecture to perform the games required by the simulation stage of MCTS will be implemented in the following work.
Chapter 5

Hardware Implementation

This chapter describes the hardware implementation of a processing element capable of playing a Chess endgame from a given starting position. This component generates all moves, evaluates them, and selects one for the side-to-move, updates the position and side to move and successively repeats these actions until a stop condition is verified.

5.1 Auto-Player Hardware Architecture

The Auto-Player hardware component plays a game of Chess for both white and black sides from a given position until a stop condition is verified. Figure 5.1 shows main blocks that were designed.

The bitboards used to encode a chess position and all control signals stored in the input and output registers are shown in tables 5.1 and 5.2. Every time a move is applied, the Output Register is updated with the new position. Once the game ends, the Output Register contains the final condition.

The Moves Generation component generates all moves available in the given position. It consists of four sub-blocks, as shown in figure 5.2.
Table 5.1: Data on the input register.

<table>
<thead>
<tr>
<th>Data</th>
<th>Size (in bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial White Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Initial Black Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Initial Pawns Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Initial Rooks Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Initial Queens Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Initial Kings Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Initial Fifty Rule Count</td>
<td>6</td>
</tr>
<tr>
<td>Initial Player’s Turn</td>
<td>1</td>
</tr>
<tr>
<td>Reset Signal</td>
<td>1</td>
</tr>
<tr>
<td>ROMs Address Reset Signal</td>
<td>1</td>
</tr>
<tr>
<td>Enable Signal</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: Data on the output register.

<table>
<thead>
<tr>
<th>Data</th>
<th>Size (in bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>White Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Black Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Pawns Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Rooks Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Queens Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Kings Bitboard</td>
<td>32</td>
</tr>
<tr>
<td>Fifty Rule Count</td>
<td>6</td>
</tr>
<tr>
<td>Player’s Turn</td>
<td>1</td>
</tr>
<tr>
<td>Game End Signal</td>
<td>1</td>
</tr>
<tr>
<td>Game State</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 5.2: Move Generation sub-blocks.

The Pseudo-Legal Move Generator produces all moves in a position without verifying beforehand if they leave the side-to-move king in check (as detailed in subsection 5.1.1). For each generated move, a target bitboard, a from bitboard, which piece-type is played, the starting and target positions are produced on this component output. Both the from and target bitboards are one-hot bitboards with, respectively, the starting and target positions marked. One new pseudo-legal move is generated per clock cycle, until all possible moves have been processed.

The Do Move performs one move on a position. The move is applied by removing the starting position from the input bitboard and adding the target position to it, as shown in figure 5.3. A full set of bitboards containing the updated game position is produced on its output, along with the information if there was a capture or if a pawn moved.

Figure 5.3: Logic operations to perform a move on the white bitboard.

The King Position sub-block is an encoder that produces the king square position (from 0 to 63) from a one-king bitboard.
The Piece Attacked sub-block (detailed in subsection 5.1.3) verifies if the King square is under attack, therefore it checks if the proposal move is illegal or not. This sub-block also checks the square where the piece of the generated move has moved to, to determine if the moved piece is under attack and if so, which is the piece-type of the attacked piece. This information is used to apply a penalty on the evaluation (as shown in algorithm 8 subsection 4.2).

The Moves Memory stores all legal moves, by saving the origin and target squares, and which is the piece type to move. This memory allows to access all legal moves after every move is processed and, if the evaluator is active for this set of moves, evaluated.

The Evaluator (detailed in subsection 5.1.2) is an hardware implementation of the evaluation algorithm used in software. This component receives a position from the Move Generator and the information if the piece moved is under attack. The evaluation for each move is then produced at the component output.

The Move Chooser component selects the move to be played (random or the best evaluated) and applies it in the input position. Once the move is performed, the new position is saved in the Output Register. Figure 5.4 shows all sub-blocks used to choose and perform a move.

![Figure 5.4: Move Chooser sub-blocks.](image)

The Best Move sub-block determines which is the best-evaluated move. Once every value from the Evaluator is processed, this component produces the memory address to access the best evaluated move.

The Random Move sub-block is a hardware implementation of the simplified random model (shown in algorithm 10 section 4.2). This sub-block contains two Read-Only Memories (ROMs) with pseudorandom pre-generated numbers. One ROM with the 0 and 1 values in the same proportion that the best-evaluated move must be chosen. This ROM contains 16384 values and it uses 0.5 BRAMs. The second ROM contains 10240 pseudo-random values of 12 bits, between 0 and 1. This ROM uses 4 BRAMs.

The Game Terminator verifies if the game reached an end. The sub-blocks used in this component are shown in figure 5.5.
The 50-Rule Counter sub-block produces a flag once the 50-Rule count reaches 50. Initially, this count is set using the value from the Input Register. The capture or pawn move informations are used to reset the count. The count value is incremented every time the Output Register is updated with a new position.

The Game End sub-block verifies if the actual game reached a checkmate, a draw (stalemate or draw by the fifty rule) or a 4-piece game situation. This sub-block contains a counter to check how many pieces are still in the game in the initial bitboard. For this, the 50-rule counter flag, the initial bitboard, the king attacked information both on the initial bitboards and the ones regarding the generated moves are inputs of this component.

---

Figure 5.5: Game Terminator sub-blocks.

Figure 5.6: Chess Auto-Player flowchart.

Figure 5.6 contains a flowchart of the circuit behaviour, demonstrating how the loop within the imple-
mented architecture works. The Auto-Player component generates all moves and saves the legal ones in a memory. If a stop condition was not reached and the evaluation is enabled for this set of moves, the Auto-Player component evaluates them. One move is selected for the side to move, the position and side to move are updated and this process is successively repeated until a stop condition is verified.

The Evaluator, Move Generator and Check Legality components, which are the most complex blocks of the circuit, are further detailed in the following subsections.

### 5.1.1 Pseudo-Legal Move Generator

The moves generation on the Pseudo-Legal Move Generator is split in the following phases:

1. Extract one bit at a time from one of the black or white bitboards, which corresponds to, selecting one of the player's pieces in order to generate its moves;

2. Generate every move for the extracted piece. For this generation, every piece on board is treated as being from the opponent and, therefore, moves that capture the player's own pieces may be generated. A new bitboard with all possible target squares is created at this phase;

3. All positions where the player's pieces reside are removed from the bitboard created in the previous phase;

4. Each possible target square is isolated in a new bitboard. Therefore, one new bitboard is created for each possible move.

Figure 5.7 demonstrates all phases described above, and exemplifies the moves generation for a white rook. The Bitboard shown in the figure is a reduced 3x3 board example which contains both white and black pieces, where the red positions are the white pieces and the green ones the black pieces. In the first phase, one piece is extracted from the white bitboard (a bitboard with only the red positions). In the second phase, the moves are generated including the ones where the piece with the same colour is captured. In the third phase, the move that captured the same colour piece is removed. In the fourth phase, the generated move bitboard with two moves is split in two one-hot bitboards.

![Figure 5.7: Moves generation phases.](image)

A Least Significant Bit (LSB) Extractor module [35], shown in figure 5.8, is used to extract the first '1' bit from a bitboard, either for extracting a piece to generate its moves (1st phase) or for splitting all
generated moves (4th phase). This module generates a new one-hot bitboard per clock cycle. Each extracted bit is removed from the input bitboard, until no more ‘1’ bits are available.

![Figure 5.8: LSB Extractor module.](image)

The block diagram circuit that performs the moves generation first three phases is shown in figure 5.9, where the LSB Extractor sub-block extracts one piece, the Position sub-block produces the positional information about the extracted piece and the 4 move generators sub-blocks generate the moves and remove the positions where pieces with the same colour were captured.

![Figure 5.9: Move generation sub-blocks.](image)

The LSB Extractor sub-block receives all bitboards and the side-to-move (Player input). This component extracts the black or white pieces, depending which is the side-to-move. Also, the component produces a valid signal, activated each time a new piece is extracted, and indicates which is the type of the extracted piece.

The Position sub-block extracts which Square, File, Rank, Diagonal and Anti-Diagonal where the extracted piece resides, using multiple encoders.

Two distinct methodologies are used to generate the moves, one for the sliding pieces and another for the non-sliding pieces. For the sliding pieces, an 8-squares sliding move generator was developed. This module uses the row occupation and piece position to produce all possible positions where the piece can move to. In the row occupation, all pieces are treated as being opponent's pieces. For the Rook, the file and the rank in which the piece resides are extracted. Then, two 8-squares sliding move generators are used to compute the possible moves for the file and rank. Once generated, all possible moves are
integrated into a 64-bit bitboard. For the Queen, the same methodology is applied, although, two more 8-squares sliding move generators are used to compute the diagonal and anti-diagonal moves. For the non-sliding pieces, the operations are applied directly on 64-bit bitboards. For this, the offset positions where the piece can move are stored. In the King case, a bitboard is created where all the surrounding piece squares are marked as possible moves. For the Pawn, two 64-bit bitboards are created, one for marking the position where the pawn advances forward and other for the diagonal captures. If there are no pieces to be captured by the pawn, the positions on the diagonal captures bitboard are removed. Then, both pawn moves bitboards are merged. Once the moves are generated, for both sliding and non-sliding pieces, all positions where the side-to-move pieces reside are removed from the moves bitboards (removing the positions where pieces of the same colour were captured).

Since the move generation creates one bitboard for each piece with all possible positions marked, each generated bitboard is split on \( n \) new bitboards, where \( n \) is the number of available moves in the

The Move Extraction circuit, shown in figure 5.10.

Figure 5.10: Move extraction circuit.

The Move Extraction circuit contains six registers to save the generated moves bitboards and piece-types of each one of them. Each player has a maximum of six pieces on board, therefore, a maximum of six moves bitboards is possible. For each valid signal, the corresponding move bitboard is saved in a new register. This process is managed by the Local Control Unit. This Control Unit also manages which is the register used by the LSB Extractor and which is the piece-type of the extracted move. Every time the register bitboard is fully processed by the LSB Extractor, a Local Reset signal is activated to reset
existing registers within the LSB Extractor and the Control Unit selects a new register. Once all loaded moves bitboards are processed by the LSB Extractor, the Extraction Ended signal is activated.

The Move Extraction has a maximum throughput of one new bitboard per clock cycle. Every time the Local Reset is activated, the data flow breaks for one clock cycle. Also, the Move Extraction circuit has a latency of two clock cycles, since one clock cycle is required to load the first register and another to extract the first one-hot Move Bitboard from the Moves Bitboard.

The Pseudo-Legal Move Generator component (move generator and move extracting circuits) has a maximum throughput of one new Move Bitboard per clock cycle and has a total latency of five clock cycles.

5.1.2 Evaluator

The Evaluator was developed to evaluate one position per clock cycle, this way not throttling the data flow. All components shown in this subsection have a latency of one clock cycle, unless otherwise stated. Also, every component has a throughput of one result per clock cycle.

Figure 5.11: White pieces evaluation circuit with the evaluation penalizing components.
Figure 5.11 shows the circuit used to evaluate the white pieces and the components used to penalize the evaluation, if the piece moved is under attack (penalty shown in algorithm 8 subsection 4.2). In what regards the black side, the evaluation the same circuit is used, removing only the evaluation penalizing components (marked with "P" in figure 5.11).

All Delay-Unit components are a set of $n$ serial connected registers used to synchronize output signals when needed, where $n$ is the number of clock cycles that the input is delayed by.

The White Rook Eval component extracts the Rooks file position, verifies if the Rook is on an open or semi-open file and evaluates the Rook piece accordingly.

The King Square Encoder extracts the Kings position. Then, the White King Table Eval evaluates the king accordingly to its position.

To evaluate the Queens, the evaluations for all possible quantities (from 0 to 4 Queens) are encoded in the Queen Eval component. The Queens quantity is retrieved from the Queen Counter component.

The Piece Attacked Value Table produces the value for the piece type if it is under attack. For this, the inputs of this component include the piece type, a bit that signals if the piece is under attack and which is side-to-move. The side-to-move input is used to select if the piece value should be added or subtracted to the white evaluation.

Signed arithmetic is used in the adders. Also, the output sizes for every adder present on the white evaluation were designed to guarantee that a result overflow never happens.

Table 5.3 shows all components that have more than one clock cycle of latency.

<table>
<thead>
<tr>
<th>Component</th>
<th>Latency (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline Pawns Extractor</td>
<td>4</td>
</tr>
<tr>
<td>Full Pawns Eval</td>
<td>7</td>
</tr>
<tr>
<td>White Rook Eval</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.3: Components latency present on the white evaluation circuit.

The full evaluation circuit has a 12-clock cycle latency, 11 cycles to evaluate each colour piece plus one cycle to perform the arithmetic operations between the white and black sides evaluations.

To evaluate Pawns with a throughput of one result per clock cycle, the Pawns Extractor was designed to extract four one-pawn bitboards per clock cycle. Each player has a maximum of four Pawns on board, therefore, a maximum of four one-Pawn bitboards is possible. The full circuit for this component is shown in figure 5.12. Four LSB Extractors are used to achieve the desired throughput. The LSB Extractors inner registers do not need reset signals between inputs, since all four register are always overwritten with new values. A Local Control Unit manages which of the LSB Extractors should reads the input bitboard and also selects the correct output when ready.
Once the Pawns bitboard is split in several one-Pawn bitboards, they are evaluated separately. Figure 5.13 shows the circuit developed to evaluate one white Pawn. The circuit is replicated for all pawns and all evaluations are added between them.

Figure 5.12: Pawns Extractor component circuit for white Pawns.

Figure 5.13: White Pawn evaluation circuit.
Firstly, the file, rank and square positions where the piece resides are extracted from the One-Pawn bitboard. These file and rank positions are used to verify if the pawn is isolated, backwards, exposed or passed, and to get the pawns count in its file. The square position is used to get the pawn positional evaluation. All evaluations for each position are stored in the Pawn Table Eval Encoder.

The Isolated Pawn Analyser checks if there is any pawn with the same colour in the adjacent files. For this, the positions to the adjacent files, for every possible file of the input piece, are saved in a memory. Figure 5.14 shows an example on a reduced 4x4 board on how is verified if a white pawn is isolated. In this example, the auxiliary bitboard contains the first and third files of a board are marked, since the input Pawn sits on the second file. Then, if the White Pawns Bitboard does not have at least one pawn in the positions marked on the Auxiliary Bitboard, the input Pawn is isolated.

The Backwards Pawn Analyser checks if there is any player’s Pawn further back in the file and adjacent files of the pawn under analysis and, if not, marks this Pawn as a backwards pawn. In this case, the Auxiliary Bitboard contains the positions for the same file and adjacent ones which are in lower ranks.

The Exposed Pawn Analyser checks if there is any pawn of the opponent side-to-move in the same file of the pawn that is being analysed. Therefore, the Auxiliary Bitboard contains the positions for the same file.

The Passed Pawn Analyser checks if there are no pawn of the opponent side-to-move in the same and adjacent files of the pawn that is being evaluated. Therefore, the Auxiliary Bitboard contains the positions for the same and adjacent files.

The File Pawn Counter calculates the number of side-to-move pawns that are in the same file of the one that is being analysed.

All the bonus or penalties on the evaluation depending on the status signals are pre-calculated and saved in a memory (Pawn Status Eval Table), replacing the arithmetic operations on the original algorithm. This approach provides a lower latency than applying arithmetic operations.

Once the status and positional evaluations are calculated, the positional evaluation and the status depending evaluation are added, if the Pawn is a passed Pawn, a value of 3 times the positional evaluation is also added.

Every component of the White Pawn evaluation circuit has a latency of one clock cycle, therefore, five clock cycles are needed to evaluate one Pawn. The full white Pawns evaluator has a total latency of five clock cycles.
seven clock cycles, since two more cycles are required to add each Pawn evaluation.

5.1.3 Piece Attacked

The Piece Attacked component verifies if a certain square is being attacked by, at least, one of the opponent's pieces and, if yes, verifies which is the player's piece type that is under attack. To perform this verification, a similar method to the one in moves generation is applied.

To verify if a non-sliding piece is under attack, a bitboard is created with all possible positions of attack by the opponent. Then, it is verified if the opponent has any piece on those positions.

![Figure 5.15: Methodology for checking a Pawn attack in a reduced board example.](image)

Figure 5.15 shows an example, in a 3x3 board, of how is checked if a piece is being attacked by an opponent's side-to-move Pawn. In the Kings case, the same method is applied.

To verify if a piece is being attacked by a sliding piece, a bitboard is created with the closest pieces in the same file, rank and diagonals (Queens case) regardless of their colour. Then, it is verified if the opponent has any sliding piece on those positions. The 8-bit sliding attack component is used to create this bitboard. This component verifies which are the pieces closest to the input piece, the only positions that a sliding piece can perform an attack. Figure 5.16 shows the methodology applied to check rook attacks, the input position is marked in red at each state. The Full Occupation Bitboard contains all black and whites pieces. In the Queens case, the same method is applied, although the diagonal and anti-diagonal are also analysed.

![Figure 5.16: Methodology for checking a Rook attack in a reduced board example.](image)

The Piece Attacked component have a latency of 1 clock cycles and a throughput of one result per
clock cycle, for both non-sliding and sliding pieces.

5.2 Conclusions

The Auto-Player hardware component is capable of generating the pseudo-legal moves for a Pawn, King, Queen or Rook and with a maximum throughput of one move per clock cycle and five clock cycles of latency. Components to perform a move in a Chess position, to compute the positional information about a piece and to check if one of the sides-to-move is attacking a certain square were designed to verify which of the generated moves are legal to perform, all these components have a throughput of one computation per clock cycle and one clock cycle of latency. So, the Auto-Player generates the moves and verifies which ones are legal to perform with a total latency of 8 clock cycles.

The Auto-Player evaluates a chess end game position with a throughput of one evaluation per clock cycle and a latency of 12 clock cycles. The evaluator performance was maximized using a high quantity of parallel computations, which was possible since each piece can be evaluated independently. The designed evaluator follows the end game evaluation of Faile where a penalty was added whenever an unprotected piece is moved into a square that can be captured.

A strategy to reduce the computation time of the evaluations is also applied in the Auto-Player, where the moves are only evaluated accordantly to a given probability. For this, the Auto-Player manages when the moves must be evaluated or when a pseudo-random move must be chosen. It contains pseudo-random pre-generated values loaded in memory to perform this management and to choose a pseudo-random move when required.

The Auto-Player performs the moves for both sides until a stop condition is verified. For this, the Auto-Player is capable of verifying when a game attains a checkmate, stalemate, draw or a 4-piece position was reached.
Chapter 6

Hardware-Software System

This chapter describes the integration of the Auto-Player component in a Hardware-Software system. The results achieved with the Hardware-Software system are compared with the Software-only approach. A parallelization method using multiple Auto-Player components is also proposed and the parallelization efficiency possible is estimated.

6.1 Architecture Integration

In the hardware architecture, 202 bits are written in the input registers and 203 bits are read from the output register, due to the reduced communication, the AXI4-Lite protocol was selected to perform all communications between the software and hardware. Since the AXI4-Lite protocol is indicated for simple, low-throughput memory-mapped communications [36]. The communications are performed through the GP ports. In this protocol, the PS is the master interface, therefore, all data transfers are initiated from the software side.

In the AXI4-Lite protocol, the data transfers have a fixed size of 32 bits. Thus, the input and output registers shown in Auto-Player component were arranged in 13 32-bit input and 13 32-bit output AXI registers. Figure 6.1 shows how the AXI registers were implemented on the Auto-Player component.
Each register has a distinct address. The information saved in the input registers and the base address offset (explained below) is shown in table 6.1. Each bitboard is saved in two AXI registers where the 32 most significant bits are saved in one and the 32 least significant bits in other. The output registers contain the same bitboards in each register, to encode the final position, and offsets as the input registers. The information saved on the AXI input and output register-12 are shown in tables 6.2 and 6.3.

<table>
<thead>
<tr>
<th>Register</th>
<th>Data</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial White Bitboard (32 MSB)</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Initial White Bitboard (32 LSB)</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Initial Black Bitboard (32 MSB)</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Initial Black Bitboard (32 LSB)</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Initial Pawns Bitboard (32 MSB)</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Initial Pawns Bitboard (32 LSB)</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>Initial Rooks Bitboard (32 MSB)</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>Initial Rooks Bitboard (32 LSB)</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>Initial Queens Bitboard (32 MSB)</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>Initial Queens Bitboard (32 LSB)</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>Initial Kings Bitboard (32 MSB)</td>
<td>40</td>
</tr>
<tr>
<td>11</td>
<td>Initial Kings Bitboard (32 LSB)</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>Multiple Data</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 6.1: Data and addresses offsets of the AXI input registers.

<table>
<thead>
<tr>
<th>Data</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Fifty Rule Count</td>
<td>5-0</td>
</tr>
<tr>
<td>Initial Player's Turn</td>
<td>6</td>
</tr>
<tr>
<td>Reset Signal</td>
<td>7</td>
</tr>
<tr>
<td>ROMs Address Reset Signal</td>
<td>8</td>
</tr>
<tr>
<td>Enable Signal</td>
<td>9</td>
</tr>
<tr>
<td>No Data</td>
<td>31-10</td>
</tr>
</tbody>
</table>

Table 6.2: Data on the AXI input register-12.

The AXI4-Lite interface consists of five different channels: Read Address Channel, Write Address Channel, Read Data Channel, Write Data Channel and Write Response Channel. Figures 6.2 and 6.3 show the channels used by AXI4-Lite. Every AXI4-Lite transition starts with a request from the PS. In a write transition, the PS transmits the address for one of the 13 registers and the data to be saved. Once the transmission is completed, a validation signal is sent from the hardware side. The address is obtained by adding the register offset (shown in table 6.1) to the registers base address, which in this case was 0x43C00000. In a read transition, the PS specifies the address for one of the 13 registers and the register data is sent by the hardware side.

<table>
<thead>
<tr>
<th>Data</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fifty Rule Count</td>
<td>5-0</td>
</tr>
<tr>
<td>Game state</td>
<td>8-6</td>
</tr>
<tr>
<td>Player's Turn</td>
<td>9</td>
</tr>
<tr>
<td>Game End Signal</td>
<td>10</td>
</tr>
<tr>
<td>No Data</td>
<td>31-11</td>
</tr>
</tbody>
</table>

Table 6.3: Data on the AXI output register-12.
In order to integrate the hardware architecture in a hardware-software system, the MCTS simulation stage algorithm was changed, as shown in algorithm 14.

**Algorithm 14** Simulation using the Auto-Player hardware component

1: procedure SIMULATION(Board[], Node.Player, Node.Fifty.Count)
2: Bitboards = Array_To_Bitboards(Board, Node.Side-To-Move, Node.Fifty.Count)
3: Load_Position(Bitboards, Node.Side-To-Move, Fifty.Count)
4: Enable_Auto-Player()
5: while Ended ≠ True do
6: [Ended, Final.Game.State] = Check_End()
7: if Final.Game.State = Tablebases then
9: Result = Probe_WDL_Tablebase(Bitboards, Final.Player)
10: else
11: Result = GameResult(Final.Game.State)
12: Rst.Auto-Player()
13: return Result
The following software functions were created to manipulate the Auto-Player component:

- Rst_Roms_Addr - resets both address counters from the random values ROMs;

- Rst_Auto-Player - resets every register, except the ones associated with the random values ROMs address counter after every simulation;

- Load_Position - writes every bitboard, which player starts and the fifty rule count on the input registers;

- Enable_Auto-Player - activates the enable signal to initiate the simulation;

- Check_End - checks if the game reached an ending and game state;

- Read_Ending_Position - reads the output bitboards and side-to-move.

To convert the board array representation to bitboards it was used the function already created to probe the tablebases. The ROMs address counters reset is only performed once before executing MCTS (one or multiple times), the others functions, explained earlier, are used every time that a MCTS simulation is required, as shown in algorithm 14.

Once the hardware architecture was integrated into the MCTS implementation, the system was implemented in the ZYBO Zynq-7010. The blocks used in this Hardware-Software system are shown in figure 6.6. This designed hardware uses a total of 16887 LUTs, 9733 FFs and 9.5 BRAMs, respectively 96%, 28% and 16% of the target platform. The achieved PL frequency is 88.9 MHz, which is equivalent to a 11.25 ns cycle duration. The PS is operated at 650 MHz. The PL clock frequency was limited due to the high utilization of LUTs. This architecture, when implemented in a Zynq-7020, using 31.74% of LUTs, reaches a clock frequency of 106 MHz. Figures 6.6 and 6.7 demonstrate the circuit layout in the Zynq-7010 and Zynq-7020.

![Figure 6.4: Circuit layout in a Zynq-7010.](image1)

![Figure 6.5: Circuit layout in a Zynq-7020.](image2)

In the following section the results obtained using the Zynq-7010 are presented.
Figure 6.6: Block diagram of the PL and PS.
6.2 Results

After completing the Hardware (Hw)-Software (Sw) system, the 30 positions tested in the Sw-only implementation were re-tested, being that the MCTS in a Hw-Sw system found also 14 moves under the same conditions, ensuring that the behaviour of the algorithm remained unchanged. Table 6.4 presents the achieved speedup for 10 positions. The pieces on board, the MCTS execution time running in a software-only approach and the MCTS execution time running in a Hw-Sw system. The full range of achieved speedups in the 30 tested positions are shown in these 10 positions. These tests were performed for 10000 MCTS iterations and 10 simulations per expansion (using O2 compiling optimizations).

<table>
<thead>
<tr>
<th>Chess Position</th>
<th>Pieces on board</th>
<th>Sw-only [s]</th>
<th>Hw/Sw System [s]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position 1</td>
<td>8</td>
<td>416</td>
<td>7.6</td>
<td>55x</td>
</tr>
<tr>
<td>Position 2</td>
<td>9</td>
<td>611</td>
<td>8.0</td>
<td>77x</td>
</tr>
<tr>
<td>Position 3</td>
<td>11</td>
<td>714</td>
<td>8.9</td>
<td>81x</td>
</tr>
<tr>
<td>Position 4</td>
<td>10</td>
<td>662</td>
<td>8.1</td>
<td>81x</td>
</tr>
<tr>
<td>Position 5</td>
<td>11</td>
<td>724</td>
<td>8.8</td>
<td>82x</td>
</tr>
<tr>
<td>Position 6</td>
<td>11</td>
<td>804</td>
<td>9.4</td>
<td>85x</td>
</tr>
<tr>
<td>Position 7</td>
<td>11</td>
<td>852</td>
<td>9.2</td>
<td>92x</td>
</tr>
<tr>
<td>Position 8</td>
<td>10</td>
<td>765</td>
<td>8.0</td>
<td>95x</td>
</tr>
<tr>
<td>Position 9</td>
<td>12</td>
<td>1024</td>
<td>10.5</td>
<td>98x</td>
</tr>
<tr>
<td>Position 10</td>
<td>12</td>
<td>988</td>
<td>10.0</td>
<td>98x</td>
</tr>
</tbody>
</table>

Table 6.4: Algorithm speed-ups achieved by the Hw/Sw system.

Table 6.4 shows that the main objective of this work was fulfilled, since the algorithm was accelerated up to 98 times using a dedicated architecture. These results show that for positions with a higher number of pieces on board, the use of a dedicated architecture achieves a greater acceleration. A larger number of pieces increases the probability of a higher number of played moves in a game, therefore, the execution time of a game increases. With this increase, the speedup achieved is bigger since more computations are performed. For the same reason, positions with more generated moves reach a higher acceleration, as for example the Position 8. In this position, Pawns are promoted sooner, leading to a bigger number of generated moves.

<table>
<thead>
<tr>
<th>MCTS Version</th>
<th>MCTS Stage</th>
<th>Selection [s]</th>
<th>Expansion [s]</th>
<th>Simulation [s]</th>
<th>Backpropagation [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sw-Only MCTS 1-Sim</td>
<td>3.6</td>
<td>1.6</td>
<td>75.3</td>
<td>1.6</td>
<td></td>
</tr>
<tr>
<td>Hw/Sw MCTS 1-Sim</td>
<td>3.6</td>
<td>1.6</td>
<td>0.4</td>
<td>1.6</td>
<td></td>
</tr>
<tr>
<td>Sw-Only MCTS 10-Sim</td>
<td>3.5</td>
<td>1.6</td>
<td>752.0</td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>Hw/Sw MCTS 10-Sim</td>
<td>3.5</td>
<td>1.6</td>
<td>4.2</td>
<td>1.7</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: MCTS stages execution time for 1 and 10 simulations per expansion.

The values presented in the table are the average execution time for the 30 positions tested. The
use of this architecture reduced the simulation stage execution time, henceforth the selection stage is the most time consuming (for 1 simulation per expansion), as shown in table 6.5. These execution times show that an higher number of simulations per expansion achieve a greater speedup of the algorithm.

6.2.1 Architecture Scalability

The developed architecture is fully scalable, this subsection proposes a parallelization method where multiple Auto-Player components are used. The Leaf parallelization method [26] (explained in subsection 3.2.1) can be applied to MCTS by using multiple Auto-Player components, as shown in figure 6.7 for two processing elements. The input registers are shared between the Auto-Player components, so each component must use distinct pseudo-random values.

The accelerations, shown in table 6.6, are calculated for the cases when 10 Auto-Player components are used for 10 simulations per expansion (1 simulation per Processing Element (PE)) and 100 simulations per expansion (10 simulation per PE). In the case of 10 simulations per expansion, the total simulation time is given by the longest of the 10 simulations, since this parallelization method requires that all parallel simulations end to start the following MCTS stage. To estimate the results for 100 simulations per expansion, the simulation stage execution time on the Sw-only (using 10 simulations per expansion) is increased by a factor of 10 and the tablebases probing execution time on the Sw-Hw system (using 1 PE and 10 simulations per expansion) is also increased by a factor of 10. This approach
would use about 166850 LUTs, 87330 FFs and 95 BRAMs therefore the system can be implemented in a Zynq-Z-7035.

<table>
<thead>
<tr>
<th>Chess Positions</th>
<th>Pieces on board</th>
<th>Simulations per expansion / (Simulations per PE)</th>
<th>10 / (1)</th>
<th>100 / (10)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sw exec. time [s]</td>
<td>Hw-Sw exec. time [s]</td>
</tr>
<tr>
<td>Position 1</td>
<td>8</td>
<td>416</td>
<td>5.6</td>
<td>75x</td>
</tr>
<tr>
<td>Position 2</td>
<td>9</td>
<td>611</td>
<td>5.4</td>
<td>115x</td>
</tr>
<tr>
<td>Position 3</td>
<td>11</td>
<td>714</td>
<td>5.9</td>
<td>124x</td>
</tr>
<tr>
<td>Position 4</td>
<td>10</td>
<td>662</td>
<td>5.2</td>
<td>129x</td>
</tr>
<tr>
<td>Position 5</td>
<td>11</td>
<td>724</td>
<td>6.0</td>
<td>123x</td>
</tr>
<tr>
<td>Position 6</td>
<td>11</td>
<td>804</td>
<td>6.3</td>
<td>130x</td>
</tr>
<tr>
<td>Position 7</td>
<td>11</td>
<td>852</td>
<td>5.8</td>
<td>150x</td>
</tr>
<tr>
<td>Position 8</td>
<td>10</td>
<td>764</td>
<td>5.0</td>
<td>155x</td>
</tr>
<tr>
<td>Position 9</td>
<td>12</td>
<td>1024</td>
<td>6.9</td>
<td>151x</td>
</tr>
<tr>
<td>Position 10</td>
<td>12</td>
<td>988</td>
<td>6.5</td>
<td>156x</td>
</tr>
</tbody>
</table>

Table 6.6: Performance increase on MCTS using 10 processing elements.

The use of a Leaf parallelization method can increase the acceleration of the algorithm, as verified by analysing table 2.3, to a maximum value of 156 times for 10 simulations per expansion and to 668 times for 100 simulations per expansion. When 1 simulation per PE is used, the selection stage becomes the MCTS bottleneck, as shown in table 6.7, affecting the maximum speedup. In this table, each MCTS stage execution time is presented for the Hw-Sw MCTS system with only one processing element, where 10 games are performed sequentially, and for a system with 10 processing elements, where each game is performed in one processing element and where 10 games are performed per processing element. These tests were performed using 10000 MCTS iterations.

<table>
<thead>
<tr>
<th>MCTS Version</th>
<th>MCTS Stage</th>
<th>Average Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Selection [s]</td>
<td>Expansion [s]</td>
</tr>
<tr>
<td>Hw-Sw MCTS 1 PE (10 sim. per exp.)</td>
<td>3.2</td>
<td>1.6</td>
</tr>
<tr>
<td>Hw-Sw MCTS 10 PE (100 sim. per exp.)</td>
<td>3.2</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Table 6.7: MCTS stages execution times and average speedup achieved.
Since only the simulation stage was accelerated using a dedicated hardware architecture, the achieved speedup increase if more simulations are performed, as shown by the results. Although the best results were estimated using 100 simulations of the same game position, the proposed architecture is easily adapted to perform simulations of different game positions, for this each PE must have distinct input registers.
Chapter 7

Conclusions

This MCTS implementation, applied in a subset of Chess, performs the selection, expansion and back-propagation stages of MCTS in software and the MCTS simulation stage in hardware. An architecture capable of playing a Chess endgame, for an input position, was developed to perform the games required by the simulation stage of MCTS, since this stage consumed about 93% of the total execution time of the algorithm. This architecture generates all moves, evaluates them, and chooses one for each player, until a stopping condition is verified. The evaluator developed in the architecture is a hardware implementation of the heuristic function applied in software, this evaluator adds the Chess specific domain knowledge to the MCTS simulation stage. The moves are evaluated according to a given probability $p$ in the developed architecture. This means that the evaluation is only calculated for $p$% of the generated moves reducing the computation time. In all other cases, a pseudo-random move is chosen. The initial goal of this work was fulfilled, since the algorithm was accelerated up to 98 times on a Zynq-7010. The use of a Leaf parallelized version with 10 processing elements, as described in subsection 6.2.1, may allow an acceleration up to 668 times. The use of Chess-specific heuristics and endgame table-bases, in the selection, expansion and simulation stages of MCTS, increased the MCTS precision and also reduced its duration by 43%. The Progressive Bias was selected as policy technique to perform MCTS selections where the developed heuristic function works as the Chess-specific domain knowledge function required by this technique.

7.1 Future Work

Firstly, a Root or Tree parallelization methods should be implemented in order to reduce the selection stage duration. These approaches will allow to perform more MCTS selections, this way balancing the time duration of the selection and simulation stages. The Root method is expected to obtain better results, as evidenced by the tests presented in table 3.3 and also this method is a simpler approach since it does not need specific access synchronization mechanisms, therefore this parallelization method should be preferred to the Tree parallelization method.

Once the selection stage is optimized, a Root+Leaf or Tree+Leaf parallelization methods can be
applied, in which the two processors of the target platform will be used and several architectures of the Auto Player will be distributed by each processor.

At last, extending this application to complete Chess games will emphasize the effect of using a dedicated architecture, due to the higher number of performed moves in complete games, since there is a higher acceleration when more moves are performed in a game. To make this modification, a knight move generator and evaluation algorithms for the initial and mid game state will be necessary to implement, both in software and in the dedicated architecture.
Bibliography


Appendix A

Tested Chess Positions

Figures A.1-A.30 show the Chess positions used as benchmark to evaluate MCTS. Table A.1 describes which side plays first and the Stockfish best move(s) for each position.
<table>
<thead>
<tr>
<th>Chess Position</th>
<th>First Turn</th>
<th>Stockfish Best Move(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>White side</td>
<td>Rc6, Rb4</td>
</tr>
<tr>
<td>2</td>
<td>White side</td>
<td>Ke3</td>
</tr>
<tr>
<td>3</td>
<td>Black side</td>
<td>Rd5</td>
</tr>
<tr>
<td>4</td>
<td>White side</td>
<td>Rd6, Rd8, Rd4, Rd5, Re7</td>
</tr>
<tr>
<td>5</td>
<td>White side</td>
<td>Kg4, Rf7</td>
</tr>
<tr>
<td>6</td>
<td>White side</td>
<td>a6</td>
</tr>
<tr>
<td>7</td>
<td>White side</td>
<td>Rg4</td>
</tr>
<tr>
<td>8</td>
<td>White side</td>
<td>Rd7</td>
</tr>
<tr>
<td>9</td>
<td>White side</td>
<td>b4, e7, a4, Re2</td>
</tr>
<tr>
<td>10</td>
<td>White side</td>
<td>b6</td>
</tr>
<tr>
<td>11</td>
<td>Black side</td>
<td>Rh4, e3, Kf6, Rf3, Kh5, Kg7, Kf7</td>
</tr>
<tr>
<td>12</td>
<td>White side</td>
<td>e6</td>
</tr>
<tr>
<td>13</td>
<td>White side</td>
<td>Rc5</td>
</tr>
<tr>
<td>14</td>
<td>White side</td>
<td>Re7, Re6</td>
</tr>
<tr>
<td>15</td>
<td>White side</td>
<td>Ra8</td>
</tr>
<tr>
<td>16</td>
<td>Black side</td>
<td>Ra4</td>
</tr>
<tr>
<td>17</td>
<td>White side</td>
<td>Rd5</td>
</tr>
<tr>
<td>18</td>
<td>Black side</td>
<td>Rg2, b4</td>
</tr>
<tr>
<td>19</td>
<td>White side</td>
<td>Rg5</td>
</tr>
<tr>
<td>20</td>
<td>Black side</td>
<td>Rf4</td>
</tr>
<tr>
<td>21</td>
<td>Black side</td>
<td>d5</td>
</tr>
<tr>
<td>22</td>
<td>White side</td>
<td>Kg2</td>
</tr>
<tr>
<td>23</td>
<td>White side</td>
<td>Rd3, Rf4, Re3, f6</td>
</tr>
<tr>
<td>24</td>
<td>White side</td>
<td>Kf3</td>
</tr>
<tr>
<td>25</td>
<td>White side</td>
<td>g4</td>
</tr>
<tr>
<td>26</td>
<td>White side</td>
<td>Re7</td>
</tr>
<tr>
<td>27</td>
<td>White side</td>
<td>Ra6</td>
</tr>
<tr>
<td>28</td>
<td>White side</td>
<td>e6</td>
</tr>
<tr>
<td>29</td>
<td>White side</td>
<td>f5</td>
</tr>
<tr>
<td>30</td>
<td>White side</td>
<td>Rb8</td>
</tr>
</tbody>
</table>

Table A.1: First turn and Stockfish best move(s) for each position.