SoC-FPGA Monte Carlo Tree Search Processor

Ricardo Rodrigues Carvalho
ricardorcarvalho@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

June 2019

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 architecture plays a Chess end game from a given starting position. 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 developed hardware-software architecture, implemented in a Zynq-7010, accelerated the MCTS execution up to 98 times. This proposed architecture, if scaled to 10 processing elements, using a leaf paralellization method, can achieve an acceleration up to 668 times. Also, the use of heuristics in the selection, expansion and simulation stage of MCTS improved the algorithm accuracy and reduced the computation time of MCTS.
Keywords: Chess, MCTS, FPGA, Hardware MCTS simulations

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. Since MCTS relies on a massive number of simulations to achieve a good solution, performing these simulations on a dedicated hardware architecture, can significantly improve the efficiency of this algorithm, which is the objective of this work.

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. 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) will be the target platform, combining efficient embedded general-purpose processing with dedicated hardware.

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 subset of Chess.

The use of heuristic functions on MCTS is explored and since the computations of heuristic functions turns 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 accordantly to the heuristic function. A strategy to reduce the computation time of the 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 will be 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 shows the available PL resources in the smallest and largest device from the Zynq-7000 family.

Device	Z-7010 (Smallest)	...	Z-7100 (Largest)
LUTs	17600	...	227400
FFs	35200		554800
BRAM	240 KB		3020 KB

Table 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.

2. Monte Carlo Tree Search Background

MCTS is a regular tree search algorithm, where the Monte Carlo method is applied to expand the tree. Monte Carlo method uses several random samplings on a certain system to learn about its outputs, that is, simulating that system, many times, with random inputs, it heuristically calculates the outputs probabilities for the inputs used on the system. In MCTS, every tree node is an outcome state of an action applied to the parent node. Every time a new node is added to the search tree, this one is simulated until it is possible to be evaluated. Since that exists many nodes 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.

In MCTS, a policy technique is used to selected a node to be added. Upper Confidence Bound for Trees (UCT) is the most popular policy technique used in the MCTS. In this policy technique the tree nodes are visited and expanded taking into account all the a priori runs of the MCTS, UCT has the following formulation:

$$UCT = \frac{w_i}{n_i} + C\sqrt{\frac{\ln(n)}{n_i}} \quad (1)$$

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. 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.

MCTS algorithm is split into the following four stages:

- Selection: The algorithm starts by selecting the root and choosing one of its children using a policy technique. The tree will be traversed by successive node selections until a non-existent child is found.
- Expansion: A new node is created and inserted in the tree with the new state corresponding to the non-existent child previously selected.
- Simulation: On the newly created state random actions (or heuristically-based) are repeatedly applied until an ending state is reached (in a game application, a win, draw, or loss state)
- Backpropagation: After simulation, the new node created is updated with the new winning and visit values. These values are added to all parents, up-to the root.

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. Variable runtime is also a big advantage of this algorithm. Since MCTS is essentially a loop of running all the four stages, it can be cut off any time. Once stopped, [11] 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 UCT.

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 corre-

sponding node still apply, and the corresponding subtree becomes the new tree.

2.1. Adding Heuristics to MCTS

The MCTS can be improved by including specific domain knowledge in the tree search. The usage of heuristic functions improves selection on MCTS, mainly, when few simulations were performed. *Progressive Bias* is an example of a possible 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:

$$ProgressiveBias = \frac{w_i}{n_i} + C\sqrt{\frac{\ln(n)}{n_i}} + \frac{H_i}{n_i + 1} \quad (2)$$

where H_i is a heuristic function that evaluates the child node i state. An edge of Progressive Bias is that, a heuristic function used in other non-MC methods can be easily adapted to be used in MCTS [2].

2.2. MCTS 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. [4] describes three MCTS parallelization methods:

The Leaf Parallelization, where there is one thread for selection, expansion and backpropagation stages of MCTS, and all other available threads are used to perform multiple MCTS simulation.

The Root parallelization, where one tree is expanded in each thread, which means that each thread develops its unique tree. Once the tree search is stopped, all the root 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.

The Tree parallelization, where each thread performs the four MCTS stages on a unique shared tree. This method requires the inclusion of specific access synchronization mechanisms.

The authors in [4] suggest that the root parallelization is a more efficient method of MCTS parallelization and they also indicate that this method prevents MCTS from remaining too long on a local optima (problem existing in UCT).

3. State of the Art

In this section, 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.

3.1. Applications Strategies Analysis

A Chess AI player [1] and MANGO (an AI Go player) [3] in 2012 and 2007, respectively, have implemented successfully their games using MCTS with UCT as policy technique. The primary focus in MANGO was to verify how much their algorithm had improved by applying several techniques (as the use of heuristic functions). In the AI Chess player [1] the primary target was to test MCTS performance when applied to Chess and examine how modifications to the base algorithm can improve it.

When heuristic functions were applied on node selection (equation 2), MANGO increased its winning rate by 33% and the AI Chess program [1] increased its winning rate by 82%. Every application used *Progressive Bias* 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, it 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). The Endgame Tablebases make it possible to quickly obtain the correct evaluation for Chess positions with only a limited number of pieces. These table are precalculated end game position evaluators generated by an exhaustive retrograde analysis.

As benchmark, MANGO played 200 games against GNU Go (3.7.10), a strong Go program. In the AI Chess player [1], round-robin tournaments of blitz Chess with five minutes clock time per side was used, where the players were several versions of itself.

Considering these implementations, it is possible to conclude that adding heuristic functions to MCTS selection becomes a huge asset. Nevertheless it must always be regarded the time consumption as adding heuristics turns MCTS slower. The usage of time-expensive heuristics, as in [3], is a good technique to reduce the computation time of MCTS.

3.2. FPGAs

This subsection presents two FPGA architectures, one implemented (figure 2) and one proposed (figure 1), both game related.

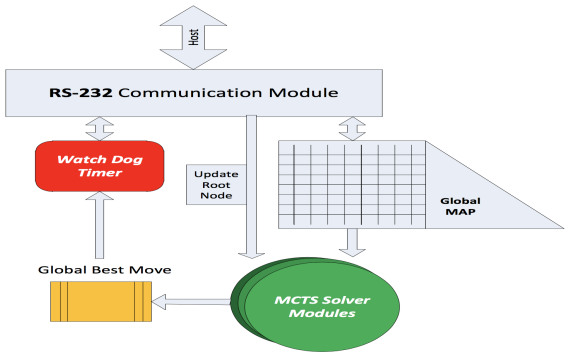

Figure 2: Implemented architecture in [8] (from [8]).

The MCTS Blokus Duo implementation in [8], uses a root parallelization and leaf parallelization methods and the architecture (figure 2) 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 news trees are created and perform MCTS on each independent Solver module.

Each MCTS solver module stores all the tree nodes in a memory. The selection, expansion, and backpropagation MCTS stages are performed in this module accessing the information stored in the memory. Each MCTS solver module contains multiple MCTS Engines that perform the simulation stage of MCTS (Leaf parallelization method).

The proposed hardware-software architecture (figure 1) in [7] is dedicated to Trax game, where pattern recognition in an image is required. In this approach, the selection, expansion and backpropagation stages of MCTS are performed in Software and the MCTS simulation stage in Hardware.

In this proposed architecture, the Universal Asynchronous Receiver-Transmitter (UART) receives a state, the move translator decodes it and sends it to the Main Processor. Then the Board Simulator gets scans the possible solutions, according to all the paths stored in the Path Memory. When the Board Simulator finds a feasible solution, this one is sent to the Solution Analyzer (SA) to be evaluated. The SA analyses the board state using multiple PRs (Pattern Recognizers) and computes a score for it. The Score Synthesizer stores the computed score if the new value is superior to the one currently stored. If a new score is saved, the Solution Register stores the best action on the Path. After that, a new next path is 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 (saved in the Solution Register) is translated and sent to the host computer through the UART.

These works show that it is possible to imple-

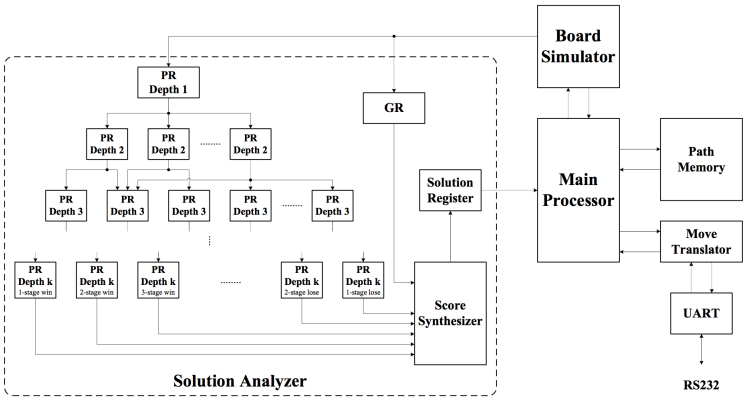

Figure 1: Proposed architecture in [7] (from [7]).

ment MCTS on a FPGA. A special care is required when the tree is stored in hardware therefore, an architecture, as the one on figure 1, in which only the simulation is performed, might be a simpler way of implementing MCTS using a hardware/software architecture.

4. MCTS Implementation

The MCTS implementation process from a software-only to a Hardware-Software system is detailed in this section. This MCTS implementation is applied to Chess endgames limited to a maximum of 1 Rook and 4 Pawns for each player.

4.1. MCTS Software Implementation

Firstly, the nodes data structure was developed (represented in listing 1). Each node contains the move performed in its parent, side to play, the information if the node is fully expanded, the number of plays, the number of wins, fifty move rule count and pointers to the its first child, first brother and parent node. The search tree structure is shown in figure 3.

```
struct {
uint8_t From_Square, Target_Square;
uint8_t Player, Fully_expanded, Fifty;
unsigned int Plays, Wins;
struct node *Child, *Brother, *Parent;
} node;
```

Listing 1: Node data structure

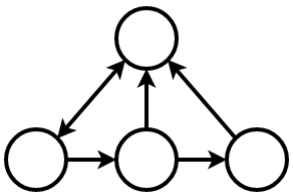

Figure 3: Tree structure .

Secondly, to apply the MCTS algorithm to Chess endgames, a moves generator and a board evaluator functions, both Chess-specific, were developed. These functions were adapted from Faile [9], an open source Chess engine. The move generator does not consider castling neither en passant pawn moves, since in end-game 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. Also, the move generation algorithm does not verify if any of the generated moves leaves the player's 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. To the Faile evaluation algorithm, a new condition was added, 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. A probing function adapted from Fathom [5], a stand-alone Syzygy probing tool, was added to this implementation, so that all 4-piece positions would be analysed by this function using the 4-men Syzygy WDL tablebases (return a Win/Draw/Loss value for a 4-piece position).

4.1.1 MCTS Stages Implementation

In the selection stage, first, the root is selected. Then, the children with the biggest Progressive Bias value (most promising children) are successively selected, until a node that is not fully expanded is reached. Once the selection ends, the selected node is expanded. To the Progressive Bias a new tunable parameter (W) was added to control how much the heuristic function affects the node selection, as shown in the following equation:

$$ProgressiveBias = \frac{w_i}{n_i} + C\sqrt{\frac{\ln(n)}{n_i}} + W\frac{H_i}{n_i + 1} \quad (3)$$

Where the evaluation function is Chess-specific heuristic H_i required by this technique. Also during the selection stage, a board array representation is updated using the moves positions in each tree node, since both the evaluator and move generation functions require a Chess position using the board array representation.

In the expansion stage, a new node is created with a distinct move from its brother nodes. The fifty rule count of the added node is updated and, if the added move is the last possible move from the generated ones, its parent node is marked as fully expanded. All possible moves are evaluated and sorted in descending order, so that the best-rated moves are expanded first. Once the node is expanded, a function verifies if the expanded node contains an ended game. For this, it verifies if the game reached a win (checkmate), draw (stalemate or by fifty move rule) or a 4-piece game was achieved. Every 4-piece position is analysed using the WDL tables. In the case of a ended game, the result is directly backpropagated, if not, the game position is simulated (one or more) times and only then backpropagated.

In the simulation stage, the expanded game position is played for both white and black sides. So, during a simulation, all moves are generated, evaluated and one is selected for the-side-to move, these actions are successively repeated until the game ends. In the case of a 4-piece game is reached, the WDL tables are used. The moves during the

MCTS Iterations	Version				
	MCTS UCT	MCTS TB	MCTS Eval	MCTS Full	MCTS Full (10 Sim)
500	2	5	7	7	10
1500	5	7	8	8	10
10000	8	8	13	13	14

Table 2: Moves found by different versions of MCTS.

MCTS Version	MCTS Stage			
	Selection [s]	Expansion [s]	Simulation [s]	Backpropagation [ms]
MCTS UCT	2.1	1.5	138	1.5
MCTS TB	2.2	1.5	102	1.6
MCTS Eval	3.2	1.6	123	1.6
MCTS Full	3.6	1.6	75	1.7

Table 3: MCTS duration (10000 iterations).

simulation are evaluated according to a given probability p (technique based on [10]. 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. Once the simulation ends, the result is backpropagaded. This technique reduces the computation time of MCTS simulations with heuristics.

In the backpropagation stage, the number of plays and wins are updated in all the parents nodes, starting from the node expanded until it reaches the root.

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

4.1.2 Results

To verify the influence of the implemented techniques in the behaviour of the algorithm, 30 Chess positions were tested. The tests were carried out for 4 versions of the MCTS, where different techniques were applied, for each version, 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 stage and simulation;
- MCTS Eval: The selection of nodes was performed using the Progressive Bias equation, the technique to reduce the computation time of MCTS simulations with heuristics applied 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 [6], 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 chosen empirically in order to maximize the precision of MCTS.

Table 2 presents the number of correct moves found by each version of the algorithm, for different numbers of iterations. The results were obtained using three simulations per expansion, since a lower number led to a large discrepancy of results for different tests of the same position. The table also shows the results obtained using 10 simulations per expansion for the strongest version.

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 average duration of each MCTS stage was also analysed for the positions tested, using one simulation per expansion, as shown in table 3.

The need of computing the heuristic function increased 36% the duration of the selection stage and 6% the expansion stage. However, it reduced the duration of simulation stage by 7%, since less moves are now performed during simulations (the random effect diminished). The Tablebases also reduced the duration 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 veri-

fied that the high duration of the simulation stage was due to the games and not due to the probing of Tablebases. 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.

4.2. 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 4 shows main blocks that were designed.

The Auto-Player saves the initial Chess position, encoded using Bitboards, the initial side-to-move and the initial fifty count in the Input Register.

The Moves Generation component produces all moves in a given position without verifying beforehand to see if they will leave the king in check . 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. Once generated, the given position is updated with each move, the King square on the updated position is extracted and it is verified if the opponent side-to-move is attacking the King square. If the King square is under attack in a generated move, that move is signalled as an illegal move (since it leaves the King in check). The Moves Generation generates the moves and verifies which ones are legal to perform with a total latency of 8 clock cycles and a maximum throughput of one move per clock cycle.

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 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 (to apply a penalty). The evaluation for each move is then produced at the component output with a throughput of one evaluation per clock cycle and a latency of 12 clock cycles.

The Move Selector 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. The Move Selector contains a memory with 0 and 1 values in the same proportion that the moves must be evaluated, so a random or the best evaluated move is selected accordantly to the value of this memory. 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. In the case of choosing the best move, this component 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. In the case of choosing a random move, this component contains other memory with pseudo-random values between 0 and 1 of 12 bits (1 integer bit and 11 fractional bits) used to select a pseudo-random move. This selection is performed by rounding the result from the multiplication between the pseudo-random value and the total number of generated moves, therefore choosing a integer value between 0 and m where m is the total number of generated moves.

The Game Terminator verifies if the game reached a checkmate, a draw (stalemate or draw by the fifty rule) or a 4-piece game situation. For this, The Game Terminator controls the 50-Rule count and once it reaches 50 ends the game by draw. Also, this component verifies if all generated moves are

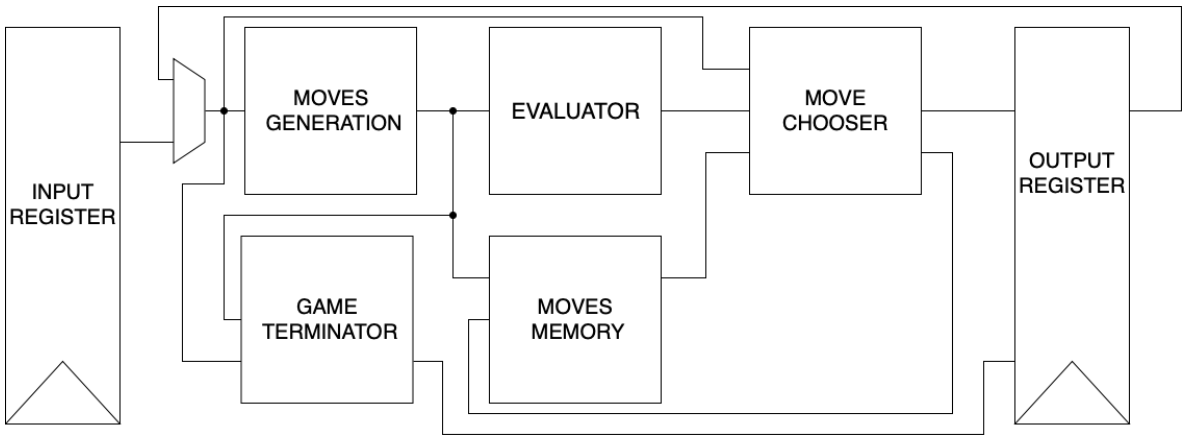

Figure 4: Chess Auto-Player hardware architecture main blocks.

illegal to perform and if so, verifies if the King is under attack in the input position (checkmate) or not (stalemate). It also contains a counter to check how many pieces are still in the game in order to stop the game once a 4-piece position is reached.

Every time a move is applied, the Output Register is updated with the new position, fifty rule count, side-to-me and game result (checkmate, draw, 4-piece or game not ended). Once the game ends, the Output Register contains the final position.

4.3. MCTS in Hardware-Software System

In this Hardware-Software system, the selection, expansion and backpropagation stages of MCTS are performed in software. While the simulation stage is performed both in hardware (simulation of games) and software (probing the WDL tables).

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 [12]. 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. Each register has a distinct address. The addresses are obtain by adding an offset to the registers base address. Each bitboard, used to encode a Chess position, 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 AXI4-Lite interface consists of five different channels: Read Address Channel, Write Address Channel, Read Data Channel, Write Data Channel and Write Response Channel. In a write transition, the PS transmits the address, through the Write Address Channel, for one of the 13 registers and the data to be saved through the Write Data Channel. Once the transmission is completed, a validation signal is sent from the hardware side through the Write Response Channel. In a read transition, the PS specifies the address, through the Read Address Channel, for one of the 13 registers and the register data is sent by the hardware side through the Read Data Channel.

In order to integrate the hardware architecture in a Hardware-Software system, the following software functions were created to manipulate the Auto-Player component:

- Rst_Roms_Addr - resets both address counters from the memories loaded with pseudo-random values;
- Rst_Auto-Player - resets every register, except the address counters of the memories loaded with pseudo-random values, after every MCTS simulation;
- Load_Position - writes every bitboard of a Chess position, which side 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 the game result;
- Read_Ending_Position - reads the output bitboards and side-to-move.

The address counters reset, from the memories loaded with pseudo-random values, is only performed once before one or multiple MCTS executions. Every time a MCTS simulation is required, the expanded Chess position is loaded in the Auto-Player input registers and the component enabled. Once the simulation ends, the result is read from the hardware side and the Auto-Player registers reset is performed. In the case where a simulation ends with a 4-piece position, the WDL tables are used to compute the position result in the software side.

Once the Hardware-Software architecture was developed, the system was implemented in the ZYBO Zynq-7010. 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.

4.3.1 Results

After completing the Hardware-Software system, the 30 positions tested in the Software implementation were re-tested, being that the MCTS, in a Hw-Sw system, also found 14 moves under the same conditions, ensuring that the behaviour of the algorithm remained unchanged. Table 4 presents the achieved speedup for 10 positions. The pieces on board, the MCTS duration running in software only and the MCTS duration 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 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

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

Table 4: Algorithm speed-ups achieved by the Hw/Sw system.

dedicated architecture achieves a greater acceleration. Since, a larger number of pieces increases the probability of a higher number of played moves in a game, therefore, the duration of a game increases. The Hardware-Software architecture reduced the simulation stage duration from 93% of the MCTS computational time to 7%, henceforth the selection stage is the most time consuming (for 1 simulation per expansion).

4.3.2 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 can be applied to MCTS by using multiple Auto-Player components, as shown in figure 1.1 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 were 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 duration on the Software (Sw)-only (using 10 simulations per expansion) is increased by a factor of 10 and the table-bases probing duration on the Sw-Hardware (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.

The use of a Leaf parallelization method can increase the acceleration of the algorithm 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. The results show that performing multiple simulations per PE is a more efficient approach than executing a low quantity of simulations per PE.

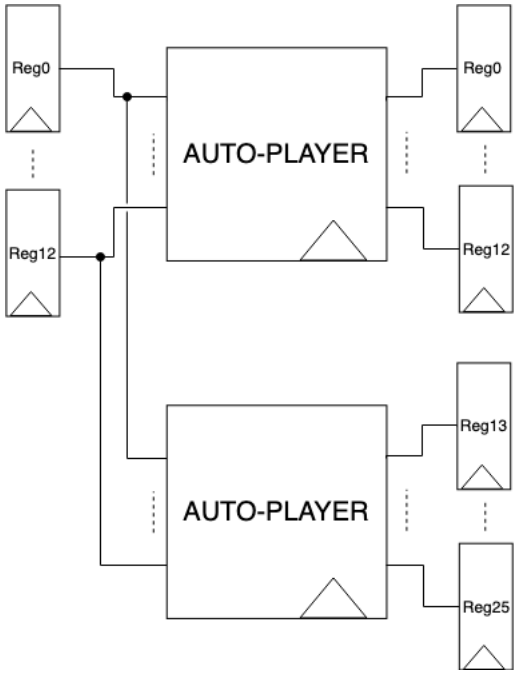

Figure 5: Leaf parallelization using Auto-Player components.

5. 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 ex-

ecution 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 may allow an acceleration up to 668 times. The use of specific Chess heuristics and endgame tablebases, 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.

5.1. Future Work

Firstly, a Root or Tree parallelization methods should be implemented in order to reduce the selection stage and tablebases probing duration. These approaches will allow to perform more MCTS selections, this way balancing the time duration of the selection and simulation stages.

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

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.

References

[1] O. Arenz. Monte Carlo Chess. (April), 2012.

[2] C. Browne and E. Powley. A survey of monte carlo tree search methods. *IEEE Transactions on Intelligence and AI in Games*, 4(1):1–49, 2012.

[3] G. M. J.-B. CHASLOT, M. H. M. WINANDS, H. J. V. D. HERIK, J. W. H. M. UITERWIJK, and B. BOUZY. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 04(03):343–357, 2008.

[4] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. Van Den Herik. Parallel Monte-Carlo tree search. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5131 LNCS:60–71, 2008.

[5] R. de Man. Fathom source code [online]. [Accessed 10 February 2018].

[6] G. Haworth and N. Hernandez. TCEC14 : the 14 th Top Chess Engine Championship. 13, 2019.

[7] Q. Lu, C. W. Sham, and F. C. Lau. An architecture-Algorithm co-design of artificial intelligence for Trax player. *2015 International Conference on Field Programmable Technology, FPT 2015*, pages 264–267, 2016.

[8] E. Qasemi, A. Samadi, M. H. Shadmehr, B. Azizian, S. Mozaffari, A. Shirian, and B. Alizadeh. Highly scalable, shared-memory, Monte-Carlo tree search based Blokus Duo Solver on FPGA. *Proceedings of the 2014 International Conference on Field-Programmable Technology, FPT 2014*, pages 370–373, 2015.

[9] A. M. Regimbald. Homepage of Faile [online]. [Accessed 10 February 2018].

[10] A. Sadybakasov. Combining Monte-Carlo Tree Search with state-of-the-art search algorithm in chess. page 37, 2016.

[11] F. C. Schadd. Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis. Master Sci:93, 2009.

[12] Xilinx and Inc. AXI Reference Guide UG761 (v13.1). Technical report, 2011.