Hardware/Software Co-Design of a Monte-Carlo Tree Search based Reversi Player

Nobutaka Kito Moeka Tsuji Kyouka Tomioka

Department of Information Engineering
Chukyo University
Toyota, Aichi, 470-0343 Japan
nkito@sist.chukyo-u.ac.jp, {T318049, T318052}@m.chukyo-u.ac.jp

Abstract— A Monte Carlo Tree Search (MCTS) based Reversi player is proposed for programmable SoCs containing both a processor core and an FPGA. It performs the tree search of MCTS with the processor core and simulates games with the FPGA by random selections of legal positions to evaluate a board in progress. The simulations are known as playouts of MCTS and consume a long time. No domain-specific knowledge other than the rules of Reversi is used for the evaluation in the player. The circuit in the FPGA is designed with high-level synthesis. It consists of processing elements for playouts and performs playouts in parallel to evaluate a given board precisely. The player was implemented for Xilinx Zynq-7000 programmable SoC on Pynq-Z1 board. The evaluation results showed that the number of playouts per second with the circuit is 2.7 times higher than that of a multi-threaded software implementation on Ryzen 7 3700X and showed that the designed Reversi player may outperform existing Reversi players.

I. Introduction

Monte Carlo tree search (MCTS) [1] is a search algorithm that combines a tree search and the Monte Carlo method. It was utilized for computer Go players, computer players of other board games, and computer players of video games. Introducing it for computer Go players is one of the major reasons why recent computer Go players are strong [1]. MCTS utilizes a playout, i.e., a simulation proceeding a game with random selections, to evaluate a state, and uses only the result of whether the simulation ends up in a favorable condition such as whether the player wins, in general. Thus, we can use it for computer players of games for which domain-specific knowledge is not known. In this paper, we design an MCTS-based player of Reversi for which domain-specific knowledge is well known, and show that the hardware-accelerated Reversi player implemented without domain-specific knowledge is not weak against existing Reversi players.

There have been several designs for realizing hardware-accelerated Reversi players. In FPT2010, an Othello competition has taken place and papers related to the competition were presented [2, 3]. In [2], a domain-specific knowledge such as mobility which is the number of legal movements was taken into account. Hardware modules for evaluating a given board and managing a board were designed. In [3], a Monte Carlo based player has been shown. The computer player was designed completely as hard logic. The difference in the number of discs between the two players at the terminal game state was used for evaluation. Recently, programmable SoCs containing both a processor core and an FPGA were developed and commercial chips are available. In [4], a hardware/software co-design of a Reversi player was shown. A Reversi player employing domain-specific heuristics was designed with high-level synthesis. Those previously proposed players used domain-specific knowledge or the pure Monte Carlo method which does not guarantee the best selection. The strength of Reversi players implemented without domain-specific knowledge or the limit of the strength of players without domain-specific knowledge is not well known.

We propose a hardware/software co-design of a Reversi player utilizing the MCTS approach for programmable SoCs. It determines the position to place a disc with the combination of tree search and playouts, and does not use domain-specific knowledge such as a well-known knowledge: the corner positions of the board are important. We can obtain the best selection with MCTS theoretically if we have enough time. Thus, we speed up the processing with hardware. We perform manual co-design for designing a Reversi player utilizing MCTS. In other words, we assign operations of the player for hardware and software manually. We assign the tree search part of the player for the processor core and assign the playout part for the FPGA in programmable SoCs. We design a circuit performing a batch of playouts in parallel with high-level synthesis to exploit parallelism in hardware.

We have implemented the design for Xilinx Zynq-7000 programmable SoC. The number of playouts per second of the designed circuit was 2.7 times higher than that of a multi-threaded software implementation on Ryzen 7 3700X processor. We evaluated the designed player by

Algorithm 1 Operations of Monte Carlo tree search

```
Generate root node v_0 and extend the node.
for time limit of the tree search do
  v_l ← Select(v_0)
  v_l ← Expand(v_l)
  Δ ← Playout(v_l)
  Backup(v_l, Δ)
end for
return the best child of v_0.
```

comparing it with existing Reversi players which may utilize domain-specific knowledge. The evaluation results showed that the implemented player may outperform those players.

This paper is organized as follows. In the next section, a brief review of Reversi and Monte-Carlo tree search (MCTS) is presented. In Section III, a hardware/software co-design of an MCTS-based Reversi player is proposed. The playout processing circuit is shown. In Section IV, the implementation details of the Reversi player, the resource utilization, and the performance of designs are shown. The comparison of the designed Reversi player with existing Reversi players is also shown. In Section V, this paper is concluded.

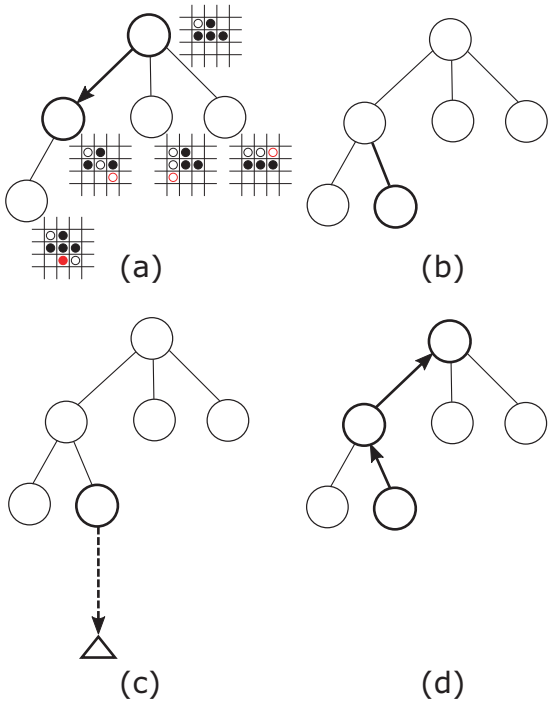

Fig. 1. Operations in MCTS approach. (a), (b), (c), and (d) correspond to the operations of *Select*, *Expand*, *Playout*, and *Backup*, respectively.

II. Preliminaries

A. Reversi

Reversi is a board game for two players. The board is divided as 8×8 grids. Players place discs on grids. Usually, one face of a disc used in the game is light and the other face is dark. Dark is assigned for the player placing the first disc and light is assigned for the other. Each player places a disc to show the face colored with the player's color. In this paper, we consider Reversi as the same game as Othello.

Traditionally, computer Reversi players are mainly designed utilizing domain-specific knowledge. It is well known among Reversi players that placing discs on four corner positions is important. During a search for the position to place a disc on computer players, boards are evaluated with some evaluation functions. Various domain-specific metrics, such as the number of legal positions, the difference in the number of discs between two players, and the sum of the predefined evaluation values associated with occupied positions, are used to evaluate a given board.

B. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) combines tree search and the Monte Carlo method. The operations of MCTS are described in Algorithm 1. As shown in the algorithm, the four operations are carried out sequentially during the assigned time for the search.

At first, a path from the root node to a node is determined as shown in Fig. 1(a). In this step, a child node is selected recursively until the most effective expandable node is reached. In the figure, as an example, we append a board of a Reversi game to each node to help understanding of the graph. A child node is added in Fig. 1(b). Then, a game from the board corresponding to the node is performed with a series of random selections of legal positions until the game ends. This operation is expressed in Fig. 1(c) and is called as a playout. The value of the playout is determined according to whether the player wins ($\Delta = 1$), draws ($\Delta = 0.5$), or loses ($\Delta = 0$). Finally, with the value of Δ, the properties of the nodes on the path are updated as shown in Fig. 1(d).

During searching a path in *Select*, the $UCB1$ value is often used to select a child node. The tree search with $UCB1$ has been named the UCT algorithm (Upper Confidence bound applied to Tree algorithm) [5]. The $UCB1$ value for child node v_j of node v is calculated as

$$\overline{X}(v_j) + c\sqrt{\frac{\ln n(v)}{n(v_j)}} \tag{1}$$

where $\overline{X}(v_j)$ is the average evaluated value of v_j through playouts, and $n(v)$ and $n(v_j)$ are the numbers how many times v and v_j are selected, respectively. The former term of (1), in other words $\overline{X}(v_j)$, intends to give priority to nodes with high winning rates and the latter term of (1)

Listing 1 Abstract top-level design of the processing circuit

```
uint PlayoutProcCircuit(board b){
  uint loop, win=0, draw=0, res, seed;

  for(loop=0; loop < N_LOOP; loop++){
    seed = SEED_CONST + loop;
    res = PlayoutPE(b, seed);
    if( res == 1 ) win++;
    if( res == 2 ) draw++;
  }

  return win + (draw<<8) + (N_LOOP<<16);
}
```

intends to give priority to nodes that are not selected enough to prevent overlooking good paths. c is a parameter to balance the two terms. If we have enough time, we can obtain the best selection with the UCT algorithm theoretically.

III. Hardware/Software Co-Design of an MCTS-based Reversi Player

In this paper, we proposed a Reversi player for programmable SoCs containing a processor core and an FPGA. At first, we show the partitioning of the player into the software part for the processor core and the hardware part for the FPGA. Then, we show the design of the hardware.

A. Structure

We consider programmable SoCs which contain a processor core and an FPGA as the target devices. The processor is suitable for sequential operations which are hard to process in parallel. The FPGA is suitable for simple operations containing parallelism. In an MCTS-based Reversi player, there are two different types of operations: the tree search and the playout. The tree search is sequential and is suitable for the processor. The playout is a simpler operation than the tree search and it is suitable for the FPGA. Therefore, we assign the tree search with the UCT algorithm for the processor core and assign the playout part for the FPGA.

B. Design of the Playout Processing Circuit

We design the playout processing circuit for the FPGA with high-level synthesis. In the MCTS-based Reversi player, the tree search program on the processor core assigns a board in progress for the playout processing circuit iteratively. Though the data size of a board is 128 bits ($= 2 \times 8 \times 8$) and is not large, transferring a board data from the processor to the FPGA consumes processing time. We designed the circuit so that it performs a batch of playouts at once to exploit parallelism in hardware and decrease overhead caused by communication between hardware and software. We obtain a reliable evaluation result Δ of a board in progress by performing a batch of playouts at once.

We show the abstract top-level design of the circuit for high-level synthesis in Listing 1. We intend that the function `PlayoutPE` in the `for` loop is realized with `N_LOOP` processing elements (PEs). We assign a different random seed for each PE to perform different random selections. We use a 16-bit linear feedback shift register to generate pseudo random numbers in a PE.

Each PE performs one playout for a request from the software. We show the steps of the playout of Reversi in Algorithm 2. *searchLegalPositions* in the algorithm searches legal positions to place a disc. It consumes a long time. It checks each of 8×8 grids to determine whether the position is legal to place a disc. In the checking of a grid, we scan 8 directions as shown in Fig. 2. We scan each

Algorithm 2 Playout of the Reversi player

Require: board b, turn T
  $cTurn \leftarrow T$
  $nP \leftarrow 1$
  while $nP > 0$ or $nPrevP > 0$ **do**
    $P \leftarrow searchLegalPositions(b, cTurn)$
    $(nP, nPrevP) \leftarrow (|P|, nP)$
    if $nP > 0$ **then**
      $pos \leftarrow P\ [\ random()\ \bmod nP\ ]$
      $placeDisc(b, pos, cTurn)$
    end if
    $cTurn \leftarrow other(cTurn)$
  end while
  if $countDisc(b, T) > countDisc(b, other(T))$ **then**
    return player T won.
  end if
  if $countDisc(b, T) = countDisc(b, other(T))$ **then**
    return player T drew.
  end if
  return player T lost.

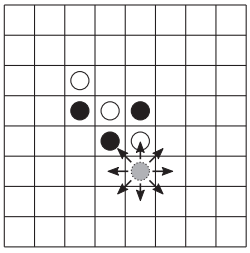

Fig. 2. Example of checking whether the position is legal. Scanning of 8 directions is performed.

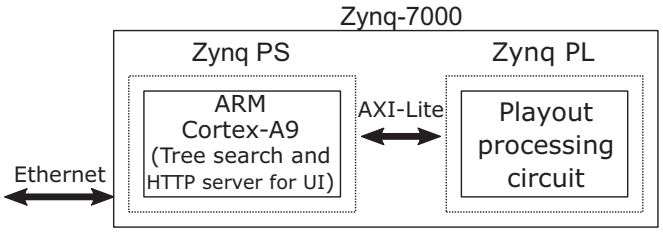

Fig. 3. Block diagram of the implementation.

TABLE I
FPGA Resource utilization of playout processing circuits (Post-implementation results).

	Available	30 faster PEs	90 slower PEs
LUT	53200	32655 (61.4 %)	47326 (89.0 %)
LUTRAM	17400	1322 (7.6 %)	962 (5.5 %)
FF	106400	36701 (34.5 %)	48169 (45.3 %)
BRAM	140	135 (96.4 %)	45 (32.1 %)

direction of the arrows for checking the grid of the gray circle in the figure. We can decrease the overall processing time by increasing circuit modules for the scanning of 8 directions.

Therefore, we can obtain various designs by changing the following parameters.

- The number of PEs.
- The number of circuit modules for the scanning of 8 directions in a PE.

We can obtain a reliable evaluation value of a board by increasing the number of PEs though the playout processing circuit consumes more resources. We can shorten the run time of a playout by increasing the number of circuit modules for the scanning of 8 directions though each PE consumes more resources.

IV. Implementation of the Reversi Player and Evaluation Results

We implemented the proposed Reversi player for Xilinx Zynq-7000 programmable SoCs [6]. We used Xilinx Zynq XC7Z020CLG400-1 on PYNQ-Z1 board. The SoC contains two ARM Cortex-A9 cores and an Artix-7 FPGA. Fig. 3 shows the block diagram of the implementation. The ARM core in the Zynq PS part performs the tree search and the Artix-7 FPGA in the Zynq PL part performs the playout part.

We used VivadoHLS 2019.2 for compiling C descriptions of the playout processing circuits and used Vivado 2019.2 for implementing them.

We designed two playout processing circuits with two different PEs. One is a design with faster PEs and the other is a design with slower PEs. The faster PE performs the detection of a legal position by scanning 8 directions in parallel. The slower PE performs the detection by scanning each direction iteratively. The slower PE is smaller than the faster PE. We populated PEs as much as possible for each design and populated 30 faster PEs for a design and 90 slower PEs for the other. The number of faster PEs was limited by the number of BRAMs. The clock frequency of both two playout processing circuits was set to 125 MHz. Table I shows the resource utilization of them.

TABLE II
Processing time of a batch of playouts from the initial boards and the number of playouts per second.

	30 faster PEs	90 slower PEs
# cycles	40,727	75,503
Time [μs]	325.8	604.0
(in real system)	(328.6)	(606.8)
Playouts per second	92,077	149,001
(in real system)	(91,308)	(148,326)

We evaluated the processing time of a batch of playouts from the initial boards in which only 4 discs are placed. Table II shows the results. The number of clock cycles was obtained with VivadoHLS and is the average of three executions. The number of playouts per second was calculated with the number of PEs, the number of cycles, and the clock period. The figures in parenthesis were observed in designed systems by running a software calling the circuits on Linux and are the average of many executions. As shown in the table, the design with the faster PEs halves the processing time compared with the design with the slower one. Note that the processing time of a batch decreases as the game proceeds.

We also evaluated the number of playouts per second of a software implementation on a computer with Ryzen 7 3700X processor and on PYNQ-Z1 board with ARM Cortex-A9 cores. The figure of a single-thread software implementation with the Ryzen processor was 18,560, and that of a software implementation, which used 90 threads for processing a batch of 90 playouts as the same as the processing of the design with 90 PEs, was 54,520. Thus, the figure of the design with 90 PEs was 2.7 times higher than that of the multi-threaded software implementation on the state-of-the-art processor. The figure of the single-thread software and the figure of the multi-threaded software with the Cortex-A9 ARM cores were 1341 and 1952, respectively. The figure of the design with 90 PEs was 76 times higher than that of the multi-threaded software on the processors in the same chip.

We described the tree search part and a user-interface part of the player in C language. We compiled the program with GCC for ARM processors. We used petalinux 2019.2 to build a Linux environment for the board. The

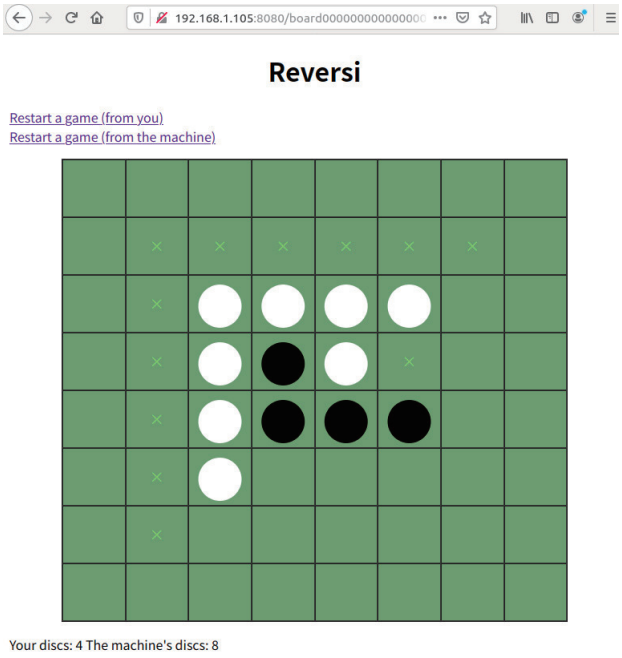

Fig. 4. User-interface of the Reversi player.

clock frequency of the ARM cores was 650 MHz. We implemented the user-interface of the Reversi player as an HTTP server generating static HTML pages for games. Fig. 4 shows the interface.

We show the result of comparing the Reversi player with existing players in Table III. The player used at most 7 seconds for each turn in the evaluation. We used Sleipnir 4 as the Web browser on Xeon E5 1620v3 processor for the existing players of [7] and [8]. We used the same processor for WZebra 4.2.4 [9] which is known as a strong player. WZebra is a customizable software. We used three settings: the default setting, with the extra large opening book provided in [9], and with the book and the search depth of "8 moves + 16 perfect moves." We compared WZebra only with the design with 90 PEs. As shown in the table, the Reversi players were stronger than the novice computer player and the computer player of hard mode though they were not stronger than existing strong players. Thus, we realized a strong-enough Reversi player with an entry-level programmable SoC chip without domain-specific knowledge of Reversi.

We can realize stronger players by longer calculation time. We compared a player with 90 PEs using 7 seconds with a player with 90 PEs using 14 seconds. The player with longer calculation time won 10 games and drew 10 games out of 20 games. The result suggests that we can realize stronger players with faster programmable SoCs.

We evaluated the number of processed batches in the design with 90 PEs during 7 seconds for deciding the first position of a disc in an new game. The average number of

TABLE III
EVALUATION RESULTS OF 10 GAMES WITH EXISTING PLAYERS.

Players	# wins (proposed - existing)	
	30 faster PEs	90 slower PEs
MathsIsFun.com [7] (hard mode)	9 - 1	10 - 0
Kuina-chan Reversi [8] (novice mode)	9 - 1	9 - 0 (1 draw)
Kuina-chan Reversi [8] (Kuina-chan mode)	1 - 9	3 - 7
WZebra 4.2.4 [9] (default setting)	—	6 - 4
WZebra 4.2.4 [9] (w/ extra large book)	—	4 - 6
WZebra 4.2.4 [9] (w/ extra large book and 8 move search)	—	0 -10

processed batches in three trials was 12,110. This means the average processing time of a pair of a batch and a tree search was 578 μs. The processing time of a batch becomes smaller than one in Table II because the tree search determines positions of some stones. The average time of a pair is also smaller than Table II. The result suggests that processing time of the tree search is small and the playout consumes a large part of the processing time in early stages of a game.

The selection of a value for c in formula (1) is important for realizing a strong player. The value was not constant in the player. We determined the calculation method of the value empirically. We used the same calculation method for both the design with 30 faster PEs and the design with 90 slower PEs. Though the design with 30 PEs was not stronger than one with 90 PEs in the evaluation, it may vary depending on the calculation method.

V. CONCLUSION

We proposed a hardware/software co-design of an MCTS-based Reversi player for programmable SoCs containing a processor core and an FPGA. We assign the tree search part of MCTS for the processor core and assign the simulation part, i.e., playouts, for the FPGA. We design a circuit performing a batch of playouts in parallel to exploit parallelism in hardware.

We implemented the Reversi players to Xilinx Zynq-7000 SoC and evaluated the Reversi players by comparing them with existing Reversi players which would utilize domain-specific knowledge. Though an entry-level chip was used in this evaluation, the Reversi players could win such players. Therefore, this approach could be reasonable for designing a strong player of games for which effective heuristics or domain-specific knowledge are not well

known.

Acknowledgements

We thank S. Sekishita, K. Terada, S. Nakata, G. Mizuno, Y. Kato, and K. Niwa for their efforts in Chukyo University.

References

[1] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol.4, no.1, pp.1–43, 2012.

[2] J. Olivito, C. Gonzlez, and J. Resano, "FPGA implementation of a strong Reversi player," *Proc. 2010 International Conference on Field-Programmable Technology (FPT)*, pp.507–510, 2010.

[3] M. Smerdis, P. Malakonakis, and A. Dollas, "CarlOthello : An FPGA-based Monte Carlo Othello player," *Proc. 2010 International Conference on Field-Programmable Technology (FPT)*, pp.515–518, 2010.

[4] P. Gangwar, S. Maurya, S. Garg, S. Goyal, A.S. Kumar, P. Dalmia, and N. Pandey, "Hardware/software co-design of a high-speed Othello solver," *Proc. 62nd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp.1223–1226, 2019.

[5] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," *European Conference on Machine Learning 2006 Lecture Notes in Computer Science*, vol.4212, pp.282–293, 2006.

[6] Xilinx, "Zynq-7000 SoC," `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`.

[7] "Mathsisfun.com - reversi," `https://www.mathsisfun.com/games/reversi.html`.

[8] "Kuina-chan board games," `https://kuina.ch/board_games/play`.

[9] G. Andersson, "WZebra," `http://www.radagast.se/othello/download.html`.