The GOmputer: Accelerating GO with FPGAs

Marco Platzner, Sven Döhre, Markus Happe, Tobias Kenter, Ulf Lorenz, Tobias Schumacher, Andre Send, Alexander Warkentin
Paderborn Center for Parallel Computing
University of Paderborn
Email: platzner@upb.de

Abstract—GO is a very popular board game, especially in the Asian world. In contrast to chess programs that are able to compete with human top players, GO programs are still rather weak. Game theory classifies GO and chess as deterministic two-person zero-sum games with perfect information, which allows to address them with game tree search techniques such as the \( \alpha/\beta \) algorithm. In principle, these games can be solved exactly. Practically, the high number of possible moves and the depth of the search tree prohibit exact solutions and require us to resort to a partial analysis of the search tree, leading to runtime consuming heuristic position evaluations.

This paper presents the GOmputer project which aims at accelerating GO through aggressively parallelized game tree search combined with FPGA-based position evaluation. We first briefly discuss the algorithmic approach for playing GO, and then focus on FPGA accelerators for several position evaluation functions. The game board is mapped as a cellular automaton directly into hardware; position evaluation functions are turned into cellular algorithms. We show the hardware implementation of several functions and report on the achieved speedups. Finally, we discuss the current state of the GOmputer project.

I. INTRODUCTION

GO is a very popular board game, especially in the Asian world, with a reported number of more than 50 million players. The typical dimension of the GO game board for tournaments is 19 \( \times \) 19, but also the smaller boards 13 \( \times \) 13 and 9 \( \times \) 9 (see Figure 3(a)) are quite common for beginners. There is only one type of stones and the opponents, i.e., the black and the white player, alternately set their stones onto the intersection points of the lines of the game board. In this paper, we denote the intersection points as fields. The players try to build chains out of their stones. A central concept in GO is the liberty which is a free field adjacent to a stone. If all liberties of a chain are occupied by the opponent, the chain is captured and all its stones are removed from the board. When both opponents pass in sequence, the game is over and the winner is determined based on features such as territorial control and captured stones. Actually, determining a winner can be far from being trivial. Typically the opponents agree on a winner, or a referee decides. For a detailed description of GO, its rules, and strategies we refer to the relevant literature, e.g., [1].

The GOmputer project at the Paderborn Center for Parallel Computing aims at developing an experimental GO machine utilizing a compute cluster and FPGA accelerators. We are interested in a parallelized and scalable GO system that can serve as a playground for research and experiments on efficiently playing and, eventually, cracking GO [2].

The GOmputer project uses the same approach than the previous, highly successful Hydra project for chess [3], [4]. The basic modules of Hydra are a game tree search algorithm, a move generator, and a position evaluation procedure. Position evaluation is a task well-suited for hardware acceleration as many evaluation functions operate on data types of only a few bits and heavily use table look-ups. Moreover, several position evaluation functions can be computed in parallel. Generating moves and searching game trees, however, are control-flow dominated tasks for which an efficient hardware mapping is not obvious. Interestingly, Hydra maps not only chess position evaluation functions to hardware but also restricted forms of move generation and game tree search.

Compared to chess, GO programs seem still to be in their infancy. Whereas chess programs are among the strongest chess playing entities of the world, GO programs do not even reach dan (master) degrees, except for very small boards. The difficulty of GO is often attributed to the high number of possible board positions which has been determined as \( \approx 10^{170} \) in comparison to only \( \approx 10^{44} \) for chess. However, the number of possible positions is not necessarily an explanation for the weakness of current GO programs (see Section II).

In the GOmputer project, we accelerate position evaluation with reconfigurable hardware, and move generation and game tree search with a compute cluster. The main contribution of this paper is the presentation of FPGA accelerators for important GO position evaluation functions. Compared to chess, position evaluation in GO is much more challenging and runtime consuming.

The paper is structured as follows: Section II introduces to the algorithmic basics of playing games such as GO. In Section III, we present the main position evaluation features used in our work. Then, Section IV discusses their FPGA implementation, followed by the evaluation in Section V. An overview over other aspects and the current state of the GOmputer project is given in Section VI. Finally, Section VII concludes the paper.

II. GO PROGRAMS

Games such as GO, chess, checkers, or Othello are deterministic two-person zero-sum games with perfect information. There are two players and the win of one player is the loss of the other. Additionally, any player has knowledge of the complete game situation at any time. Such games are
classically modeled with game trees and analyzed using the minimax principle.

Figure 1 shows an example for a game tree. The root of the tree corresponds to the current game situation and the player who has to move next, also called the max player, seeks to maximize the reward. Now, all possible moves are considered which leads to a number of new game situations corresponding to the next level of nodes in the game tree. This goes on until the leaves of the tree are reached. The leaves model game end situations which allow to determine the winner. Each end situation is assigned a numerical value which expresses the reward for the max player. In Figure 1, the leaves receive values between minus three and seven. At each max node, the max player will always decide for a move that maximizes the reward. At the min nodes, the opponent or min player will always decide for a move that minimizes the reward for the max player. The value of the root, finally, is called the minimax value and denotes the solution of the game. The minimax value equals the reward of the max player, provided that both players play consistently their best strategy.

An improvement over the minimax technique is provided by the family of \( \alpha/\beta \) search algorithms [5]. These algorithms apply pruning techniques based on two bounds, a lower bound \( \alpha \) on the reward the max player will get at least, and an upper bound \( \beta \) on the reward the min player will allow at most. Pruning reduces the number of nodes that need to be visited.

In the example of Figure 1 we have searched down to the leaves. In games such as GO or chess, theoretically, we can also generate the complete search tree, assign the values 1 for win, 0 for a loss, and 0.5 for a draw to the leaves, and then apply \( \alpha/\beta \) search to compute the minimax value of the game. In practice, there is no way to do that as the search trees for those games grow extremely large.

The typical approach for game tree search is to generate a subtree up to some depth and to assign heuristic values to the leaves of the subtree as shown in Figure 2. The heuristic values assigned to the artificial leaves form predictions about the solution of the game and will be denoted as position evaluation features for the rest of this paper. More precisely, the heuristics are often split into positional features and material features such as the number of captured stones.

The heuristic values are propagated up to the root by a game tree search algorithm to determine the best move for the max player. Real game tree search algorithms include many further improvements. For example, the search depth is typically not the same for all nodes. Rather, heuristics are used to decide where to search deeper. Then, instead of using a pure depth-first search a move generator first lists all successor nodes and sorts them according to heuristics to increase the efficiency of \( \alpha/\beta \) pruning. For example, moves that have led to cut-offs in the search tree earlier might be considered first.

The experience with game tree search shows that the actually searched subtree acts as an error filter [3]. Although the evaluation of the subtree’s leaves is based on heuristics and thus includes errors (imperfect predictions), the subtree attenuates these errors. The deeper the searched subtree is, the better is the error filter. Consequently, an efficient game program relies on both clever position evaluation heuristics and deep search. In other words, game-specific knowledge and speed matter.

In summary, a typical program for solving a deterministic two-person zero-sum game with perfect information consists of an \( \alpha/\beta \) search procedure, a move generator, and a heuristic position evaluator [5]. Although this algorithmic approach works very well for games like chess, checkers, and Othello, it did not yet lead to a successful GO program. There might be two main reasons for that. First, developing an efficient position evaluation procedure for GO is still considered a highly demanding task [6]. There is no broad consent which set of features to use and how to weight them. Second, the known position evaluation procedures are computationally highly demanding. Consequently, GO programs with heuristic evaluation functions are rather slow in terms of searched nodes per second which leads to small search depths. An error-correcting mechanism – as known from other games – could thus not be observed yet.

One specific long-term goal of the GOmputer project is to speed up the heuristic position evaluation to be able to experiment with increased search depths. A recently investigated alternative approach to playing GO are Monte-Carlo methods. For small game boards such as \( 9 \times 9 \), Monte-Carlo based tree search algorithms have been shown to be very strong. GO implementations mainly based on the technique presented in [7] and on the UCT algorithm [8] have already reached dan level. The crucial point with these algorithms is that the accuracy of a special Monte-Carlo procedure can be improved with game tree search. Moreover, these algorithms need no heuristic position evaluation procedure because the Monte-Carlo sample games are played to the end. However, this approach is extremely unlikely to scale to larger GO boards such as \( 19 \times 19 \).

### III. Position Evaluation

This section presents the heuristic position evaluation currently used in the GOmputer. In contrast to many GO programs that apply a multitude of features, we rely on only three techniques: the influence value, the analysis of dragons, and the analysis of eye spaces.
A. Influence Value

The influence value gives an indication of territorial control for a given board position and is a central position evaluation feature in GO. A position for a $d \times d$ game board is defined by the state of its $d^2$ fields, where a field at column $i$ and row $j$ is either empty or occupied by a black or white stone, i.e., $\text{field}_{i,j} = \{\emptyset, b, w\}$. The initial influence values of the fields are set as follows:

$$I^0_{i,j} = \begin{cases} +1 : & \text{field}_{i,j} = b \\ -1 : & \text{field}_{i,j} = w \\ 0 : & \text{field}_{i,j} = \emptyset \end{cases}$$

Then, the influence is computed in iterations. In each iteration, the influence of an empty field is updated by plus one for each neighbor field in the 4-neighborhood with a positive influence value, and by minus one for each neighbor in the 4-neighborhood with a negative influence value. Formally, the influence of $\text{field}_{i,j}$ after $n$ iterations is determined by:

$$I^n_{i,j} = I^{n-1}_{i,j} + \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} \text{sgn}(I^{n-1}_{k,l})$$

Fields occupied with stones do not change their influence values. After $n$ iterations, the influence values of all empty fields are weighted and summed up to the resulting overall influence. Weighting is used to emphasize certain positions on the board stronger than others. For example, since controlling the corners of the board is strategically important – at least in the beginning phase of a game – the fields near the corners are typically emphasized. The weights for the individual fields are stored in the so-called piece square table $\text{PST}_{i,j}$. The overall influence after iteration $n$ is given as:

$$I^n = \sum_{i=1}^{d} \sum_{j=1}^{d} \left( \left\{ \text{PST}_{i,j} : I^n_{i,j} = 0 \right\} \cup \left\{ \text{field}_{i,j} = \emptyset \right\} \right)$$

The numerical value of the influence corresponds to the strength of the dominating player, positive values point to the black player as being dominant, negative values to the white player. Figure 3 presents an example with the board position given in Figure 3(a) and the initial influence values in Figure 3(b). Black has a positional advantage as it has one stone more on the board than white. Figure 3(c) shows the situation after the first iteration. Assuming equal weights of one for all fields, i.e., $\text{PST}_{i,j} = 1$, ∀$i,j = 1 \ldots d$, the resulting overall influence is $I^1 = 0$, expressing that black and white are on par. In the next iterations, the overall influence changes to $I^2 = 0$, $I^3 = -11$ and $I^4 = -30$, pointing to a clear advantage for white. The influence values after the fourth iteration are shown in Figure 3(d).

B. Dragon Analysis

Worms, dragons, liberties, and eye spaces are very important concepts in GO. A worm is a group of stones of one color connected in 4-neighborhood. Figure 4 shows four such worms, two black and two white ones. A dragon consists of one or more worms which are grouped such that they cannot be dissected by the opponent. While an exact analysis of dragons in GO can be quite involved, we resort to following heuristic to identify them: if two stones of two different worms are
in 8-neighborhood (diagonal), we check the horizontal and vertical fields encapsulating the diagonal. If both fields are empty, the worms form a dragon structure. This way, we might miss some dragons but all structures identified as dragons are actually dragons. The two black worms and the two white worms in Figure 4 are classified as dragons by our heuristic. The diagonal stones connecting the worms are indicated by dashed lines.

The feature dragons evaluates how strong the dragons of a color are or to what extend they are in danger, respectively. The computation of this feature requires a number of steps and uses a set of parameters which are listed in Table I.

We start with the concept of dragon strength. The strength of dragon $i$ is determined by the weighted number of border fields

$$
\text{strength}_i = \text{border}_i \cdot p_0,
$$

where border is the sum of all fields in 4-neighborhood of dragon stones that are either empty or occupied by the opponent. The white dragon in Figure 4, for example, has a border of 11. Note that larger dragons are stronger than smaller ones, and lengthy dragons are stronger than clustered ones.

The next concept is that of liberties, which are an indicator for the dragon’s ability to survive and grow. A liberty of first order, liberty$^1_i$, is an empty field in 4-neighborhood to a stone of a dragon. A liberty of $k$-th order, liberty$^k_i$ is an empty field in 4-neighborhood to a liberty of order $k - 1$ that has not already been labeled as liberty. In Figure 4, the liberties of order one to four of the white dragon are indicated. Capturing single stones, worms, or dragons requires to take all the first order liberties from these structures. Therefore the first order liberties indicate the ability to survive, whereas the higher order liberties point to the potential for growing the dragon and responding to attacks. The number of orders considered can vary between GO implementations. In our work, we consider liberties up to the 4-th order [9]. Thus, the total liberty of dragon $i$ is computed by summing up the weighted liberties of all orders:

$$
\text{liberty}_i = \sum_{k=1}^{4} \text{liberties}_i^k \cdot p_k
$$

TABLE I

<table>
<thead>
<tr>
<th>parameter</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_0$</td>
<td>weight for border fields</td>
</tr>
<tr>
<td>$p_1$</td>
<td>weight for 1st-order liberties</td>
</tr>
<tr>
<td>$p_2$</td>
<td>weight for 2nd-order liberties</td>
</tr>
<tr>
<td>$p_3$</td>
<td>weight for 3rd-order liberties</td>
</tr>
<tr>
<td>$p_4$</td>
<td>weight for 4th-order liberties</td>
</tr>
<tr>
<td>$p_5$</td>
<td>weight for a safe eye space</td>
</tr>
<tr>
<td>$p_6$</td>
<td>maximum liberty value</td>
</tr>
<tr>
<td>$p_7$</td>
<td>weight for stones of a dragon</td>
</tr>
</tbody>
</table>

The analysis of dragons also includes safe eye spaces. Safe eye spaces are empty areas within dragons, in which the opponent cannot build live structures anymore (see Section III-C). A live structure is a structure that can be made unbeatable. A dragon with two such save eye spaces cannot be captured and will ultimately survive. We add parameter $p_5$ to liberty$^i$ if there is one safe eye space attached to dragon $i$. In case the dragon contains two or more safe eye spaces, the liberties are of no more interest and liberty$^i$ is set to a predefined maximum value $p_6$.

Now we compute the dragon weakness. We consider a dragon as weak, if it has an insufficient number of liberties or safe eye spaces. In case a dragon is captured, we model the loss as sum of two components, the number of stones and the strength. With stones$^i$ denoting the number of dragon stones, the resulting equation for the weakness is:

$$
\text{weakness}_i = \max \left(p_6 - \text{liberty}_i, 0\right) \cdot (\text{stones}_i \cdot p_7 + \text{strength}_i)
$$

In a sense, the first factor which is in $[0. . . 1]$ models the certainty of getting captured, while the second factor expresses the loss. Finally, the dragon evaluation feature considers the complete set of dragons $D$ of one color, sums up all dragon strengths and subtracts the sum of all dragon weaknesses:

$$
\text{dragons} = \sum_{i=1}^{\mid D \mid} (\text{strength}_i - \text{weakness}_i)
$$

In the software implementation, the recognition of dragons is done recursively. Starting with a field of the target color, a new dragon is created and assigned an identifier. Then, all neighbors are visited and labeled with the current dragon identifier if they have the same color and adhere to the 8-neighborhood rule discussed above. The recursion terminates when the visited field has already been considered earlier, is occupied by the opponent, or empty. The border and liberties of dragons are computed similarly.

C. Eye Space Analysis

The analysis of eye spaces is a feature on its own but also a prerequisite for the evaluation of dragons (see Section III-B). After all dragons have been identified, we search for eye spaces which are basically areas surrounded by a dragon and, optionally, the border of the game board. We differentiate between three types of eye spaces, basic eye spaces, safe eye spaces, and very safe eye spaces. Note that these terms are
our definitions used for the eye space evaluation feature and are not necessarily identical with GO literature.

In general, an eye space surrounded by a dragon might contain empty fields and structures of the opponent. Similarly to dragon analysis, the identification of eye space candidates is also based on a recursive search, this time differentiating only between the dragon’s color and other fields. If such an eye space is found, it is marked as basic eye space. A basic eye space can be promoted to a safe eye space if it meets several conditions with respect to its size and position on the board, i.e., whether it is located at a corner, at the border, or in the middle of the board. These conditions are common GO knowledge. A newly identified safe eye space is attached to the surrounding dragon. If the dragon already had one safe eye space attached, both eye spaces are classified as very safe. If the dragon already had more than one safe eye space attached, the newly identified eye space is also turned into a very safe eye space.

An example is shown in Figure 5. Consider the black dragon in the bottom-left corner of the board. Together with the border of the board it surrounds an eye space of size eight. Now, it is known that an eye space in the corner of the board is only guaranteed to be safe if its size is less than eight. The eye space in Figure 5 is not safe which is demonstrated by the white dragon inside the eye space. The white dragon contains itself two safe eye spaces and, thus, cannot be captured. Another example is shown at the top of the board in Figure 5. An eye space located at the border of the board is known to be safe only if its size is less than 10. In the example shown this is the case. White is not able to build any live structure within this eye space.

For each eye space \( l \in S \), the number of fields contained, \( \text{size}_l \) is noted. A basic eye space is evaluated as

\[
\text{eyespace}_l = \text{size}_l \cdot k(\text{size}_l) \cdot \left( p_b + p_{pst} \cdot \sum_{(i,j) \in \text{area}_l} \text{PST}_{i,j} \right).
\]

In this equation, \( \text{area}_l \) denotes the set of all fields of the eye space and \( k() \) represents a weighting function depending on the size of the eye space. Such a function is motivated by the fact that small eye spaces are more difficult to attack than larger ones. \( p_b \) and \( p_{pst} \) are constant weighting parameters and \( \text{PST} \) is the piece square table introduced in Section III-A. Using the piece square table for the evaluation of eye spaces allows to emphasize eye spaces in strategically important areas of the board. For safe and very safe eye spaces, we add corresponding bonus parameters, i.e., \( \text{eyespace}_s = \text{size}_s \cdot k(\text{size}_s) \cdot (p_s + f_1) \) for safe eye spaces and \( \text{eyespace}_v = \text{size}_v \cdot k(\text{size}_v) \cdot (p_v + p_s + f_1) \) for very safe eye spaces. Finally, all single eye space evaluations are summed up:

\[
\text{eyespaces} = \sum_{i=1}^{|S|} \text{eyespace}_i
\]

D. Positional Value

The value assigned to a certain position equals the weighted sum of the influence and the differences between black and white in dragons, eye spaces and captured stones:

\[
\text{position} = \text{influence} \cdot p_i + (\text{dragons}_b - \text{dragons}_w) \cdot p_d + (\text{eyespaces}_b - \text{eyespaces}_w) \cdot p_e + (\text{captured}_b - \text{captured}_w) \cdot p_c
\]

A positive positional value puts black at the advantage, a negative value white. Additionally, we have implemented a function for determining the winner in case the game reaches an end. The function includes algorithms for territorial analysis, removal of dead structures, and seki analysis (situations where stones of both colors share first order liberties).

IV. FPGA ACCELERATION

In this section, we present the hardware design of the GO position evaluation features. We start with the representation of the game board in hardware, followed by the influence, dragon and eye space analysis. Then, we describe two novel features, the Euler number and small eye spaces, which have been developed in the light of limited FPGA logic resources.

A. Game Board in Hardware

The central data structure for the position evaluation algorithms is the game board. Typically, GO boards are quadratic of dimensions \( d \times d \). In hardware, we implement the game board as cellular automaton. Each cell represents a field on the game board that can be empty or occupied by a black or white stone. For symmetry reasons, we augment the cellular automaton with border cells. Figure 6 outlines the cellular automaton. The game board is parameterizable and contains

\[
\begin{align*}
\text{parameters used for eye space evaluation} \\
\hline
\text{parameter} & \mid \text{usage} \\
\hline
p_b & \text{weight for basic eye space} \\
p_s & \text{weight for safe eye space} \\
p_v & \text{weight for very safe eye space} \\
p_{pst} & \text{weight for piece square table} \\
k() & \text{weight depending on the size of an eye space} \\
\hline
\end{align*}
\]
overall $d^2 + 4d + 4$ cells, out of which $4d + 4$ are border cells. The cells of the automaton are connected in an 8-neighborhood, except the border cells which have connections as shown in Figure 6. Each cell is assigned an index from $[1 \ldots d^2]$ running from top-left to bottom-right.

![Fig. 6. GO game board as cellular automaton](image)

**B. Influence**

The influence is computed as discussed in Section III-A. For this algorithm, each cell contains a color, an influence, and a $pst$ register. The color register is initialized with empty, black, or white. The influence register is initialized to $+1$ for black fields, $-1$ for white fields, and zero for empty fields. The $pst$ register is initialized with the value of the piece square table for the corresponding field.

Then, all cells of the automaton compute the influence iterations in parallel. In each step, a cell reads the color and influence values of the adjacent cells in 4-neighborhood and increments its influence register by one for each neighbor with positive influence, and decrements its influence register by one for each neighbor with negative influence. One such step corresponds to one iteration and takes four clock cycles. In our experiments, we have computed four iterations which takes 16 clock cycles. We have also developed a version that takes much more logic and computes an iteration in only one clock cycle. In our experiments, however, we have used the slower version as we have been resource-limited.

The next step is multiplying the local influence values with the elements of the piece square table, which is also done in parallel. To simplify this step, we have chosen the parameters of the piece square table as powers of two. The final step is adding all values, for which we have developed again two versions. One version exploits more parallelism using adder trees, and the second version performs the addition sequentially in $d^2$ cycles.

**C. Dragons and Eye Spaces**

The identification of connected components is a necessary precondition for the analysis of dragons and eye spaces. In the cellular automaton, all techniques that need to identify connected components base on one central technique, the propagation of identifiers. Identifier propagation assigns a unique identifier to each connected component, using either 4-neighborhood or 8-neighborhood. The basic variant of the algorithm requires each cell to contain the color register, the index register, and an identifier register. The identifier is initially set to equal the index.

The identifier propagation algorithm runs in iterations. In each iteration, each cell sends its identifier together with its color to each of the neighbors in 4 or 8-neighborhood. Hence, each cell receives four or eight identifier/color pairs. Out of these data, a cell considers only those from neighbors matching its own color. Now the cell computes an updated identifier by taking the minimum out of all the identifiers of neighbors with matching color and its own identifier value. Eventually, this leads to the propagation of the minimum identifiers within connected components.

![Fig. 7. Propagation of identifiers](image)

**Identifer propagation works slightly different for dragons.** Here, we have to use an 8-neighborhood. In determining the minimum over all identifiers from neighbors of the same color, data from diagonal cells are only considered if the horizontal and vertical neighbors enclosing the diagonal are empty fields. As an example, in Figure 7(a) the cells with initial identifiers 67 and 77 are in 8-neighborhood. After some iteration steps, cell 67 receives the identifier 47. One step later, cell 77 follows and takes on the identifier 47 since the adjacent horizontal (cell 76) and vertical (cell 68) neighbors are empty fields. Figure 7(b) shows the result of the dragon identification.

The propagation of identifiers exploits the parallelism in the cellular automaton. The runtime of the algorithm depends on the actual number and form of the connected components. An
upper bound on the number of steps is \( \left\lceil \frac{d}{2} \right\rceil \cdot d + \left\lfloor \frac{d}{2} \right\rfloor \), which would be required for a snake-like dragon spreading over the complete board. In our design, we catch the termination by adding a flag to each cell that is raised when the cell did not update its identifier in the current step. The flags from all cells are and-ed together to form the global termination signal.

The identifier propagation technique is the basis for both dragon and eye space analysis. According to Section III-B, first the borders and liberties of dragons are determined by identifier propagation, and then strengths, weaknesses, and the overall dragon evaluation feature are computed. Since the parameters are powers of two, the required division can be implemented by a shift operation.

Also the eye spaces are identified by identifier propagation and then classified into basic, safe and very safe eye spaces as discussed in Section III-C. The analysis of dragons and eye spaces leads to huge logic requirements, especially if we try to implement each step in the algorithms in one clock cycle. To be able to utilize the FPGAs available in our computing environment (see Section V), we have reduced the exploited parallelism in these hardware accelerators. For example, during the identifier propagation step we compute the four directions (west, east, south, north) sequentially instead of parallel.

D. Euler Number

The Euler number is a concept known from image processing and denotes the difference between the number of objects and holes in a binary image. A binary image contains foreground pixels (object pixels) and background pixels (potentially hole pixels). An object is a maximally large set of object pixels in 8-neighborhood. A hole is a maximally large set of background pixels in 4-neighborhood. However, such a set of background pixels is not classified as a hole if it is situated in a row. The number of runs in row \( i \) (see, for example, \([10]\)) is a sequence of object pixels \( R(i) \times \) pixel patterns in the image to be extended by a leading background pixel. Two runs in adjacent rows are neighbor runs if there is at least one pixel in each run which is in 8-neighborhood to the other pixel. To check for such neighbor runs, a certain set of patterns is moved over two adjacent rows. The number of neighbor runs between row \( i \) and \( i - 1 \) is denoted as \( O(i) \).

The sequential algorithm scans the image row by row. The first row cannot contain any holes. Thus, the Euler number of an image consisting of one row, \( I_1 \), is given by the number of objects, \( E(I_1) = R(1) \). Considering the next row, we have to add the number of runs as they are potentially new objects, and to subtract the number of neighbor runs, as they denote extensions of previous objects. The resulting Euler number of an image with two rows is therefore \( E(I_2) = E(I_1) + R(2) - O(2) = R(1) + R(2) - O(2) \). For the overall image with \( n \) rows, we get:

\[
E(I_n) = \sum_{i=1}^{n} R(i) - \sum_{i=2}^{n} O(i)
\]

Figure 8 shows an example for a binary image \( I_6 \). The number of runs are \( R(i) = \{1, 2, 1, 1, 3, 1\} \) and the number of neighbor runs are \( O(i) = \{0, 2, 2, 0, 2, 2\} \) for \( i = 1 \ldots 6 \), which results in the Euler number \( E(I_6) = 9 - 8 = 1 \), denoting that this image contains one object more than holes.

This algorithm lends itself to a parallel implementation as \( R(i) \) and \( O(i) \) can be computed independently for all rows or pairs of rows, respectively. In \([11]\), a hardware implementation for computing the Euler number was shown, using a pipelined architecture for computing runs and neighbor runs. We have adopted this architecture to compute the Euler number for a GO board position. Here, the Euler number is the difference between the number of dragons and the eye spaces of some side. Obviously, it is desirable to have a few larger dragons with many eye spaces rather than many small dragons with only a few eye spaces. Hence, a small Euler number is preferable. The contribution of the Euler number to the overall positional value is thus defined as \((Euler_w - Euler_b) \cdot p_i\), where \( p_i \) is a weighting parameter. To our knowledge, such a feature has not been used yet in GO position evaluation.

Our implementation scans all cells for a match with certain patterns of the 8-neighborhood. The neighbors that need to be checked are shown in Figure 9(a). Overall, we check for 14 different patterns that, if found, contribute with a local Euler number. Figure 9(b) displays one specific pattern. If the cell to the north-west of an occupied cell is of the same color and the fields to the north and west are empty, the resulting local Euler number is 2, independent of the cells to the south and east. In the figure, these cells are denoted as don’t cares (dc). Figures 9(c) and 9(d) show two more patterns, differing only in the north-west field. If this field is empty, the Euler number is -2; if the field is occupied by some stone the Euler number is 0. At this point, our Euler number algorithm actually differs from its original form as we are considering cells (pixels) of three different colors (black, white, and empty).

E. Small Eye Spaces

Small eye spaces is a simplified variant of eye space analysis that considers only eye spaces of size one. Such small eye spaces do not only play an important role in GO, but are also straight-forward to identify as we have to check only the
8-neighborhood of empty cells. The algorithm proceeds like the full eye space analysis and first generates all dragons by identifier propagation as discussed in Section IV-C. Then, the small eye spaces are identified and attached to surrounding dragons. We differentiate between dragons without any eye space of size one, dragons with exactly one eye space of size one, and dragons with more than one eye space of size one. The small eye spaces are weighted and added to the positional value as shown in Section III-C.

F. Accelerator Generation and Hardware/Software Interface

A cell of the game board contains a number of modules, depending on the features to be evaluated. By adding and removing modules we can easily generate hardware layouts for different combinations of position evaluation features. The VHDL code is parameterized with the dimensions of the game board. Depending on the actual dimensions, the identifiers and some other signals are binary encoded. For example, for \( d = 19 \) we need 9 bit per identifier. The color information, on the other hand, does not depend on the dimension and is encoded with 2 bits.

Besides having local connectivity, all cells of the game board are also connected to an I/O controller module. The I/O controller itself connects via a 64-bit wide FPGA-PCI bridge to the host CPU and establishes the data transfers. To that end, the I/O controller implements different communication functions. For example, the I/O controller can collect data from the single cells or other modules and partition it into 64-bit chunks for sending it over the PCI bus.

The GO program running on the host CPU is provided with a number of functions for position evaluation. The functions are listed in Table III and are exactly the same, independent of whether the position evaluation is computed in software or accelerated with FPGA hardware. Before an evaluation is started, the board position has to be transferred. This can be done with sendBoard() which transmits the complete board, or incrementally with doMove() and undoMove(). The latter functions transfer also data for several cells at once to fully utilize the 64-bit CPU/FPGA interface. If the addition of one stone removes the last liberty from an opponent’s structure, the complete structure is captured and removed from the board. In such a case, the corresponding call of the doMove() function includes all changed fields. The same applies to the undoMove() function. For a \( 19 \times 19 \) board, typically 12 doMove() operations correspond to one sendBoard() operation.

At the beginning of a game and whenever the game tree search algorithm starts computing a new subtree, sendBoard() will be used. However, most of the time we will evaluate sequences of boards within subtrees, for which the more efficient incremental functions are used.

V. Results

The FPGA accelerators for the position evaluation features have been coded in VHDL and synthesized with Synplicity’s Synplify and Xilinx ISE to the XC2VP70-5 device mounted on the AlphaData ADM-XP FPGA boards used in our compute cluster. We have conducted several synthesis experiments and evaluations with different combinations of position evaluation features and board dimensions.

The first accelerator version includes the features influence, dragon analysis, and eye space analysis as described in Section III. This version comes with huge resource requirements and allowed us to scale up only to boards of dimension \( 9 \times 9 \) on the given FPGA. The detailed resource requirements are shown in Table IV. As can be seen, the accelerator for the influence feature is not very resource-demanding, even for a \( 19 \times 19 \) board it takes only 38% of the FPGA’s logic resources. The analysis of dragons and eye spaces, however, leads to the enormous resource demand. We have further synthesized the accelerators for dragon and eye space analysis to the currently largest available Xilinx FPGA, the Virtex5 LX330. There, we could scale up to a \( 13 \times 13 \) board – a game board of size \( 19 \times 19 \) still requires a device with 1.8 times the capacity of the Virtex5 LX330.

Table V presents the performance of this first accelerator version for the \( 9 \times 9 \) game board. The FPGA design was clocked at 33 MHz. To estimate the performance of the sendBoard() version of transferring data to the accelerator, we have sent and evaluated 100’000 complete board positions.

---

**Table III**

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendBoard()</td>
<td>writes complete board to the accelerator</td>
</tr>
<tr>
<td>doMove()</td>
<td>adds one or more stones to the board</td>
</tr>
<tr>
<td>undoMove()</td>
<td>removes one or more stones from the board</td>
</tr>
<tr>
<td>setParameters()</td>
<td>writes parameters for evaluation features</td>
</tr>
<tr>
<td>sendPST()</td>
<td>writes piece square table</td>
</tr>
<tr>
<td>getEvaluation()</td>
<td>compute positional value (callback function)</td>
</tr>
<tr>
<td>getDetailedEvaluation()</td>
<td>includes internal states for debugging</td>
</tr>
</tbody>
</table>

**Table IV**

<table>
<thead>
<tr>
<th>board size</th>
<th>logic utilization [% of XC2VP70]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 7 \times 7 )</td>
<td>6%</td>
</tr>
<tr>
<td>( 9 \times 9 )</td>
<td>9%</td>
</tr>
<tr>
<td>( 11 \times 11 )</td>
<td>13%</td>
</tr>
<tr>
<td>( 13 \times 13 )</td>
<td>18%</td>
</tr>
<tr>
<td>( 15 \times 15 )</td>
<td>24%</td>
</tr>
<tr>
<td>( 17 \times 17 )</td>
<td>30%</td>
</tr>
<tr>
<td>( 19 \times 19 )</td>
<td>38%</td>
</tr>
</tbody>
</table>
To estimate the performance with doMove()/undoMove(), we have transferred start positions with sendBoard(), followed by randomly generated incremental positions transferred with doMove(). Thus, successive boards differ only in one field. Besides performance testing, the results of the software and hardware versions have also been compared successfully for functional equivalence. The software versions of the position evaluation features have been executed on a 64-bit Xeon processor clocked at 3.2 GHz. In software, the performance differences between the sendBoard() and doMove() versions are vanishingly small. The differences in execution time have been measured to be at most one percent. Hence, in Table V we report the performance only for the sendBoard() version. The table presents the execution time, the throughput in board evaluations per second, and the attained speedup which ranges up to 3.80. The difference between the sendBoard() and doMove() versions in hardware are rather small for the given board dimensions and position evaluation functions.

Table IV also shows the modest resource requirements for the Euler number evaluation feature. Euler number and influence easily fit together on one FPGA, even for a board size of 19 × 19. Table VI shows the resulting performance data for an accelerator computing these two features. Again, the FPGA has been clocked at 33 MHz. This time we have executed 10 million positions for both, the software and the hardware version. We achieve a speedup of about 3.55 using sendBoard(), and 17.89 using doMove(). The accelerator designs for influence and Euler number exploit sufficient parallelism. Hence, the performance is limited by the communication overhead. This can be seen by the improvement for the doMove() version over the sendBoard() version in Table VI.

The next accelerator version uses the features influence, Euler number and small eye spaces for a 19 × 19 board. Although Table IV shows that the sum of the resource requirements for these features exceeds the device capacity, we have successfully implemented this accelerator on one FPGA, as the features share quite some hardware structures.

For the evaluation, we have again clocked the FPGA at 33 MHz and tested 100’000 board positions. The feature small eye spaces has not yet been integrated into the software version. Hence, Table VII does not compare exactly the same algorithms in software and hardware. The software techniques shown in this table are definitely more sophisticated. Consequently, we do not report speedup figures for this experiment.

The achieved throughputs and speedups are summarized in Figure 10. While we have been resource-limited, the figures presented give a good impression of the range of attainable speedups. Given the growth in logic capacity, we will soon have available large enough FPGAs to implement dragon and eye space analysis also for 19 × 19 boards and realize higher speedups. Moreover, during accelerator design we have restrained ourselves from exploiting more parallelism in order to reduce logic requirements. Larger devices will allow us to exploit this huge potential and achieve even higher speedups.

VI. THE GOMPUTER PROJECT

The GOMputer is being realized and tested on the Arminius cluster of the Paderborn Center for Parallel Computing [12]. This cluster comprises computing nodes with dual 64-bit Xeon 3.2 GHz / 4 GB RAM and a 10 GBit/s Infiniband interconnect. Some of the nodes are equipped with FPGA boards Alpha Data ADM-XP/PCI. The FPGAs are connected via a 64-bit wide FPGA-PCI bridge for data transfers, clocked at 66 MHz.

A. Parallel Game Tree Search and Move Generation

Most of the implemented techniques for the sequential and parallel implementations of game tree search and move generation follow the work of [13], [3] on parallelizing chess programs. Our move generator heuristically sorts possible moves...
The parallelized $\alpha/\beta$ algorithm relies on a work-stealing approach which utilizes dynamic worker/employer relationships. One processor is the master and starts the tree search as if there were no other processors. The other processors send requests for work to randomly chosen processors in the cluster. A busy processor receiving such a request checks whether it is able to share some of its work. In that case, it will send back a work package. Otherwise, a message is sent back informing the requesting processor that there is no work available. When a worker finishes a work package, it sends the result back to its employer which resolves the worker/employer relationship. The worker will then, again, send a request for work to a randomly chosen employer. A worker/employer relationship can also be resolved by an employer. This occurs, for example, when the employer receives or produces an $\alpha/\beta$ cut-off in its search tree. In this situation some workers are known to compute parts of the search tree which are of no more interest. The parallel search procedure has been enhanced with the help of distributed transposition, history, and killer tables [13].

The parallelized game tree search has been evaluated for boards of size $9 \times 9$ and $19 \times 19$ on cluster configurations with up to 128 processors. The speedups achieved depend strongly on the search depth. For example, for a $19 \times 19$ board and search depth 3, we achieve a maximum speedup of about 10 with 16 processors. For the same board but with search depth 4, the maximum speedup is about 6 using 32 processors. A general observation is that larger board sizes and deeper searches give more opportunity to share work. On the other hand, there is also more potential for $\alpha/\beta$ cut-offs. The development of improved work balancing techniques to efficiently employ more than 16 processors with $\alpha/\beta$ search is subject of future work.

B. Graphical User Interface

The graphical user interface is a completely separated module written in Java. Besides providing the user interface for playing GO, the GUI can display various statistics which facilitates debugging and game analysis. Games can also be saved, loaded, and resumed. Figure 11 displays a screen shot of the GUI. The GUI connects per ssh to the cluster and communicates using the GO text protocol (GTP). There are several playing modes, human vs. human, human vs. engine, and engine vs. engine, where engine can be the GOmputer or any other GO program running GTP (e.g., GnuGO).

C. Current Status

Besides the development of the basic algorithms and accelerated implementations for move generation, game tree search, and position evaluation, a good GO machine requires extensive parameter tuning and optimization. The GOmputer project has recently entered this parameter calibration phase.

Overall the position evaluation contains 91 parameters, out of which 56 are elements of the piece square table (using symmetries). Finding reasonable values for these parameters is hard as many of the parameters are related and different phases of a game need different parameter settings. We are addressing this problem with a genetic optimizer utilizing a dan game data base to determine the fitness of a current parameter set. We have selected 50 board positions randomly out of dan games, and generated all possible moves for each position. Then, we have evaluated these moves with the current parameter set and ranked them according to their score. The fitness of the current parameter set is set to equal the rank of the actual move of the dan player in our sorted list of moves. Consequently, we try to tune the GOmputer to mimic moves from dan players. We have to note that this approach considers only single moves out of a game and, specifically, does not look at strategies which are realized by a sequence of moves.

While we have generated initial parameter sets with a few days of optimization in order to perform test games, we are currently still in the process of optimizing. Some first remarkable results, which also match the opinion of many GO players, are that the weighting parameters for 4th-order liberties and larger eye spaces are typically set to zero. That is, these features are not considered important at all.

We have also conducted a few test games with a GOmputer setup using two nodes (4 processors and four FPGA boards). As position evaluation features, we have used dragon and eye space analysis in software, and influence, Euler, and small eye spaces in hardware. We have played on a $9 \times 9$ board against a 1 dan, a 1 kyu and a 14 kyu player. We won against the 14 kyu with a handicap of 5 stones, and lost the other games. Further, we have won against GnuGO regularly on the very small board size of $7 \times 7$. GnuGO is ranked 5 kyu on some GO servers currently.

The main result from this first round of tests is that the GOmputer makes highly reasonable and strategically important moves during the first 50 moves of the game. This was also attested by the human opponents. After that, the strength of our machine drops dramatically. One specific comment of the human opponents was that the GOmputer often failed to
attack the opponent’s structures. This lack of aggressiveness is likely due to suboptimal parameter tuning. Our initial parameter set uses, for example, rather high weights for the influence feature during all phases of the games. As the influence feature is most important for the first phase of a game, we need to deemphasize the weight of this feature in the course of the game or switch it off totally in later phases. It must also be noted that our current GOmputer uses a rather small number of evaluation features compared to existing GO programs. Here we have to test more extensively which features are really of value.

VII. Conclusion

We have presented the design and implementation of reconfigurable hardware accelerators for certain GO position evaluation features. Although we have been constrained by the limited logic capacity of the used FPGA device, we have achieved reasonable speedups. Importantly, there is plenty of room for improvements. Once larger devices are available, we can more aggressively exploit parallelism. In parallel, the designs can be further optimized to increase the clock rate. Further, CPU-FPGA interfaces with lower latency, such as HyperTransport or PCI Express, will also help increasing the speedup. Another option we have not explored yet is the use of multiple FPGAs to evaluate one board position.

In this paper, we have also provided an overview over the GOmputer project including the parallelized game tree search. Here as well there is room for further research addressing the efficiency of the load distribution, scalability, and combinations of game tree search with recent Monte-Carlo approaches. The current playing ability of the GOmputer is that of a novice. We need to boost the strength of the GOmputer which requires primarily parameter tuning and the use of adaptive weights. Finally, we want to investigate how the performance measured in searched nodes per second affects the playing strength.

VIII. Acknowledgment

The contributions of following people are acknowledged: Moritz Bödicker, Ingo Döpke, Marcus Dürksen, Alexander Gebel, Dennis Hannwacker, Nicolas Heine, Martin Klose, Matthias Linnemann, Tobias Leckeband, Christian Nacke, Albert Römer, Lars Schäfers, Jens Simon, and Gunnar Steinert.

REFERENCES