Rotated bitboards in FUSc#

Johannes Buchner

10th August 2005

Abstract

There exist several techniques for representing the chess board inside the computer in chess programms. The straight-forward-approach of just maintaining an array representing the 64 squares on a chessboard works fine, but has several drawbacks in move-generation as well as evaluation, which are frequently used in modern chess programs. Thus, the chess programming community has developed more advanced board representations, one of them beeing the bitboard representation. The basic idea is based on the observation that modern CPUs are 64bit-processors, i.e. the length of a word in machine language is often 64bit, which corresponds to the 64 squares on the chess board. The advantage of this representation lies in the availibility of very fast bit-manipulating operations (like AND, OR, NOT) on modern CPUs. It is therefor possible to construct very efficient chess programms on the basis of the bitboard-approach, because, roughly speaking, the CPU operates on all 64bit "in parallel". In order to use the bitboard representation for an efficient move generation for sliding pieces, a refinement called "rotated bitboards" is needed. In this paper the basic ideas as well as the concrete implementation of the rotated-bitboards move generator in our chess program FUSc# will be discussed. In the first part the concepts of the bitboard-representation of the chess board, the advantage of bitboards in move generation and the reasons why rotated bitboards are needed for the generation of moves for sliding pieces are explained. In the second part, the concrete implementation of the move-generator in FUSc# is discussed, i.e. the ideas of part one can be studied in practice. Part three explains a technique how to verify the move-generator of a chess programm with the "perft"-command - after the general concept is described, it is used to show that the move-generator of FUSc# works 100% correct. Finally, in an outlook to the prospects of further research, it is discussed in how far the rotated-bitboards representation of the chess board will profit of the availibility of 64bit-processors (AMD64, IA64), and what future enhanchments could be done.

Contents

1 Introduction to bitboards

1.1 The basic idea of bitboard representations

The idea for the bitboard representation of the chessboard is based on the observation that modern CPUs are 64bit-processors, i.e. the length of a word in machine language is nowadays often 64bit. Those 64bit-words will correspond to the 64 squares on the chess board, and those "bitboards" (the name that is used for an unsigned int64) are used to represent various information about the position on the chessboard. The advantage of this representation lies in the availibility of very fast bit-manipulating operations on modern CPUs: On 64bit-machines, operations like AND, OR, NOT etc. can be executed on a 64bit "bitboard" in only one cycle. It is therefor to construct very efficient chess programms on the basis of the bitboard-approach, because, roughly speaking, the CPU operates on all 64bit "in parallel". Details how this works exactly will be given in sections 1.2 and 1.3.

In the history of computer chess, there were several authors who used variants of the bitboard-representation in their chess engines. As early as in the seventies, Slate and Atkin described the idea of using bitboards in their program "CHESS 4.5" (see [3], chapter 4). Another prominent programms that used this technique successfully is the former computer chess champion "Cray Blitz", written by Robert Hyatt, who continued to develop the program as an open-source project called "Crafty" ([9]). Another world-class chess engine using bitboards is DarkThought, developed at the university of Karlsruhe in the late 90s. Crafty and DarkThought were also the first programs that used an important refinement of the bitboard-representation called "rotated bitboards" (see section 1.4) . The author of DarkThought Ernst A. Heinz gives an overview of rotated bitboards as used in DarkThought (see [2]), which inspired much of our own developments.

1.2 Bitboards to represent a chess positions

In each bitboard, a special information/property of the position can be encoded, where a "1" in the bitboard means the property is true for the given square, while a "0" means the property is not true. As an example, consider a bitboard "w_occ" that contains the information which square is occupied by a white piece - all squares corresponding to a "1" are occupied by a white piece, the others are not.

In order to represent a chess position, one "bitboard" is of course not enough - only the combination of several bitboards can contain the complete information of a position. Let's consider the following bitboards:

- one bitboard for each type of piece: "pawns", "knights", "bishops", "rooks", "queens", "kings"
- two bitboards "w_occ" and "b_occ" indicating which squares are occupied by which color
- a collection of bitboards encoding the occupied squares in a "rotated" manner (see 1.4)

In this representation, the white pawns can be obtained by "ANDing" the "pawns"-bitboard (in which the pawns of both colors are encoded) with the "w_occ"-bitboard:

```
white_pawns = pawns AND w_occ
```

Another example is computing the empty squares. For this, the white and black pieces are "ANDed", and then the bitwise complement ("NOT") is formed:

```
empty_squares = NOT (w_occ AND b_occ)
```

By following this idea and using the bitwise operations "AND", "OR", "NOT" etc., many more interesting information can be computed from the bitboards very efficiently.

1.3 The bitboard-approach towards move-generation

The move-generation is used many times during the search-algorithms used in chess programs. Therfor, an efficient move-generation is needed. Based on the bitboard-approach, there exist different strategies for each of the piece-types in chess. One important concept is to compute bitboards of all possible moves (e.g. of a knight) from all the squares beforehand during the initialisation of the program, and store this information in a data-structure that provides efficient access to these pre-computed moves during the move-generation. For non-sliding pieces, this approach works straighforward, but for sliding-pieces some more tricks are needed. which are explained in the next section (1.4). But let's start with looking at generating moves for pawns, which uses a different but very elegant way of using the bitboard-representation.

1.3.1 Pawns

The idea for generating pawn moves using bitboards is based on the "shift"-operations that exist on all microprocessors: by shifting the bitboard containing the white pawns to the left by 8 positions, the non-capturing moves of all (up to 8) white pawns can be generated simultaneously (this shifted bitboard has to be "ANDed" with the empty_squares in order to be valid)! For pawn captures, just shift to the left by 7 and 9 respectively, and "AND" with the black pieces. Although this looks amazingly fast on first sight, in practice some of the advantage of the parallel generation is lost when the moves must be put into a move list seperately (see section 2.2 for details). Maybe this could be avoided in some cases, and there are some ideas for future developments (see 4).

1.3.2 Non-sliding pieces

For non-sliding pieces like knight or king all possible moves from all the squares of a chess board are computed during the initialisation of the program and stored in arrays indexed by the from-field, i.e. there exist 2 arrays:

- knight_moves[from-field]
- king_moves[from-field]

In knight_moves[c1], for example, a bitboard that contains all possible "to-squares" for a knight standing on field "c1" is stored. During move generation, this bitboard can be "ANDed" with a bitboard containing the fields that are not occupied by own pieces (i.e. NOT(own_pieces)) to produce all knight moves from c1. But there are more possibilities: if knight_moves[c1] is "ANDed" with the opponents pieces, only capture-moves will be produced (this is needed very often e.g. during quiescence search). In general, very advanced move generation schemes are possible, e.g. "moves that attack the region of the opponent king" could be genrated by "ANDing" the possible to-squares with a bitboard that encode the fields near to the opponent king. These examples show the flexibility of the bitboard-approach.

Although this technique works fine for non-sliding pieces, there are diffculties when starting to think about sliding pieces, which will be covered in the next section.

1.3.3 Sliding pieces

Computing "all possible moves from all squares" for sliding pieces is not as easy as for non-sliding pieces, because the possible moves for a sliding piece will depend on the configuration of the line/file/diagonal it is standing. For example, on a compleatly empty chessboard, a bishop standing in one of the corners will have plenty of moves, but in other positions with own pieces standing next to it and blocking its diagonal, there might be not even one move possible for the bishop. Therfor, the idea for bitboard-move-generation for sliding pieces is to compute all the possible moves for all squares **and** all configurations of the involved ranks/files/diagonals! For example, the rank-moves for a rook standing on a1 on an otherwise empty chessboard will be stored in "rank_moves[a1][00000001]", with the second index of the array beeing the configuration of the involved rank (i.e. 8 bits, with only “a1” beeing occupied as the rook is standing there itself). This works fine for rank-moves, because the necessary 8 bits for the respective rank can be easily obtained from the bitboard of the occupied pieces (this bitboard consists of 8 byte, and each of those corresponds to one rank). For file-moves of rooks and queens, and especially for the diagonal moves of bishops and queens, things turn out to be much more difficult: the necessary bits about the respective files/diagonals are spread all over the “occupied”-bitboard, they are not “in order”, as they are for rank-moves. Here the idea of rotated bitboards helps out.

1.4 Rotated bitboards

The idea of rotated bitboards is to store the bitboards that represents the “occupied squares” not only in the “normal” way, but also in a “rotated” manner. Therfor, the necessary bits representing files/diagonals are “in-order” in those rotated bitboards, as needed by the move-generation (see previous section). The “rotated bitboards” are updated incrementally during the search, i.e. when a move is done or undone. The following bitboards are maintained:

- board.occ, which represents the occupied squares in the”normal” representation
- board.occ_l90, the board flipped by 90° (for file moves)
- board.occ_a1h8, for diagonal moves in the direction of the a1h8-diagonal
- board.occ_a8h1, for diagonal moves in the direction of the a8h1-diagonal

A detailed description how these bitboards are used during move-generation can be found in sections 2.4.1 and 2.4.2. See the Appendix (5) for details about how the different rotated bitboards look like.

2 Move generation in FUSc#

FUSc# is the chess program developed by the “AG Schachprogrammierung” at the Free University in Berlin ([5]). It is written in C# and runs on the Microsoft .NET Framework ([11]). This section aims to give explain in details how the move generator of FUSc# works. At first an overview of move generation in FUSc# is given, then the generation of moves for the different pieces is explained. As explained in section 1.3, there are three main categories of piece-types for move generation:

1. Pawns
2. Non-sliding pieces (knight, king)
3. Sliding pieces (bishop, rook, queen)

For each of these categories, the steps involved in the move-generator of FUSc# are explained and illustrated by some snippets from the source-code of FUSc#. However, for these explainations, not the latest (fine-tuned, and therefor quite unreadable) version of the source-code of the FUSc#-move-generator will be used, but an earlier version where the concepts involved can be seen much clearer. Those concepts of course still form the basis of the move-generator of the latest versions of FUSc# and DarkFUSc# (our new engine, both can be downloaded from [6])

2.1 Overview of move generation in FUSc#

We will discuss now in detail how the move generator in FUSc# works. We will only deal with “movegen_w” that generates moves for white - there is a symmetrical routine for black, which is based on the same ideas and will not be treated here. The call for “movegen_w” is:

```
int movegen_w(Move[] movelist, ulong from_squares, ulong to_squares)
```

You can see that movegen_w expects 3 parameters:

- a “movelist” to store the generated moves in
- a bitboard (ulong is 64bit in .NET!) of “from_squares”, which is normally “board.w_occ”, i.e. all white pieces
- a bitboard of “to_squares”, which is normally “~board.w_occ”, i.e. the complement of all white pieces, but could also be e.g. “board.b_occ” to generate only capture moves

In the following sections we will discuss the move generation for the different pieces in detail.

2.2 Pawns

2.2.1 Non-capturing moves

Here is the code-snippet from the FUSc#-move-generator that generates (one step) non-capturing pawn moves for white:

```
1  // WHITE PAWNS (one step)
2  pawn_fields_empty = ( (board.pawns & from_squares) << 8) & (~board.occ.ll);
3  tos = pawn_fields_empty & to_squares;
4  froms = tos >> 8;
5  while (from = GET_LSB(froms))
6  {
7  board.w_attacks |= from;
8  movelist[movenr].from = from;
9  movelist[movenr].to = GET_LSB(tos);
10 movelist[movenr].det.ll = 0;
11 movelist[movenr].det.ail.piece = PAWN;
12 movelist[movenr].det.ail.flags = 0;
13 movenr++;
14 CLEAR_LSB(tos);
15 CLEAR_LSB(froms);
16 };
```

In line 2, the idea is to compute a bitboard of all the “empty squares in front of white pawns”. To get this, the pawns (standing on the “from_squares”) are shifted to the left by 8 bits, and the result is “ANDed” with the complement of the “occupied” squares (found in “board.occ.ll”). Then, this “pawn_fields_empty” is “ANDed” with the “to_squares” in order to get the destination squares (“tos”) for all the desired moves. The from-squares (“froms”) for those moves can be obtained by shifting back the “tos” by again 8 bits. After line 4, all one-step non-capturing pawn moves (that origin from “from_squares” and head to “to_squares”) have been genrated and are encoded in the two bitboards “froms” and “tos”. In the while-loop in lines 5-16 those moved are put into the movelist indivudally. Therfor, the individual moves that correspond to the bits in the bitboard “froms” and “tos” must be obtained one-by-one. In line 5, the “Least Significant Bit” (LSB) of “froms” is extracted and saved in the bitboard “from”, and in line 9 the same is done for the LSB in “tos” (it is saved in the bitboard “to”). These two bitboards, together with some additional information (like the piece that is moving) is then saved in the movelist (lines 7-12). In lines 13 and 14, the “Least Significant Bits” of “froms” and “tos” are cleared, as this was the move that has just been processed. If there is are bits left in “froms”, then the next iteration of the while-loop will extract them, otherwise the generation of one-step non-capturing pawn moves is finished.

Two-step non-capturing pawn moves are generated similarily.

2.2.2 Pawn captures

Here is the code-snippet from the FUSc#-move-generator for generating white pawn captures (to the right side):

```
   1  // WHITE PAWNS (captures right)
   2  tos = ( (board.pawns & NOT_RIGHT_EDGE & from_squares) << 9) & board.b_occ &
to_squares;
   3  froms = tos >> 9;
   4  while (from = GET_LSB(froms))
   5  {
   6  board.w_attacks |= from;
   7  movelist[movenr].from = from;
   8  movelist[movenr].to = GET_LSB(tos);
   9  movelist[movenr].det.ll = 0;
   10 movelist[movenr].det.ail.piece = PAWN;
   11 movelist[movenr].det.ail.flags |= NORMAL_CAPTURE;
   12 movenr++;
   13 CLEAR_LSB(tos);
   14 CLEAR_LSB(froms);
   15 };
```

Pawn captures are generated similar to non-capturing pawn moves, although there are some differences: in the code-snippet above, the pawn-captures "to the right side" are generated, that's why the pawns standing on the right edge may not be included in the bitboard that is shifted (this time by 9 positions) in order to generate the moves. To achieve this, a bitboard the bitboard "NOT_RIGHT_EDGE" is "ANDed" to the pawns before the shift (line 2). Another difference is that now the destination squares must contain black pieces, as we generate pawn captures. Lines 4-15 work like the respective lines for pawn non-capturing moves.

2.2.3 Special moves

To explain the section for en-passent and promotion moves lies beyond the scope of this article. The techniques used base on the concepts introduced in the previous two paragraphs about pawn moves. If you are interested in the concrete implementation, you can download the source of FUSc# at our homepage ([5]) and have a look yourself!

2.3 Non-sliding pieces

2.3.1 Knights

Here the a code-snippet from the FUSc#-move-generator that generates moves for the white knight:

```
   1  // WHITE KNIGHT
   2  froms = board.knights & from_squares;
   3  while (from = GET_LSB(froms))
   4  {
   5  from_nr = get_LSB_nr(from);
   6  tos = knight_moves[from_nr] & to_squares;
   7  while (to = GET_LSB(tos))
   8  {
   9  board.w_attacks |= from;
   10 movelist[movenr].from = from;
   11 movelist[movenr].to = to;
   12 movelist[movenr].det.ll = 0;
   13 movelist[movenr].det.ail.piece = KNIGHT;
   14 movelist[movenr].det.ail.from_nr = from_nr;
   15 movelist[movenr].det.ail.flags |= FROM_NR_COMPUTED;
   16 if (board.b_occ & to) movelist[movenr].det.ail.flags |= NORMAL_CAPTURE;
   17 movenr++;
   18 CLEAR_LSB(tos);
   19 };
   20 CLEAR_LSB(froms);
   21 };
```

Generating moves for the white knight starts in line 2, where "board.knights" (containing the knights of both colors) is "ANDed" with the "from_squares" (which normally contain all the white pieces). The result (a bitboard containing the white knights) is saved in "froms". In line 3 the LSB of "froms" is extracted and saved in the bitboard "from", which then only contains one bit set (at the position where the first white knight resides). Then, in line 5, the number of the bit set in "from" is computed by the routine "get_LSB_nr(from)" and saved in "from_nr". The "from_nr" is needed to index the array "knight_moves" in line 6 (this array contains all ever possible knight moves from the square that is given as index, see section 1.3.2). The destination squares for knight-moves ("tos") are computed by "ANDing" the "knight_moves[from_nr]" with the "to_squares", which could be all empty squares or all black pieces, e.g., if only the generation of certain types of moves is desired (capturing/non-capturing). After that, the generated moves are put in the movelist in lines 7-21.

2.3.2 Kings

The move-generation for the king is analog to the move-generation for kinghts, using "king_moves[from_nr]" instead of "knight_moves[from_nr], of course. Like in many other engines, assuring that the king is not left in check after a move is not done inside the move-generator, but inside the search-routine, i.e. the FUSc#-move-generator produces only pseudo-legal moves that possibly leave the own king in check.

2.4 Sliding pieces

2.4.1 Rooks

Here the a code-snippet from the FUSc#-move-generator that generates moves for the white rook:

```
   1 // WHITE ROOK
   2 froms = board.rooks & from_squares;
   3 while (from = GET_LSB(froms))
   4 {
   5 from_nr = get_LSB_nr(from);
   6 rank_pattern = board.occ.byte[from_nr >> 3];
   7 file_pattern = board.occ_l90.byte[l90_to_normal[from_nr] >> 3];
   8 tos = (rank_moves[from_nr][rank_pattern] | file_moves[from_nr][file_pattern])
& to_squares;
   9 while (to = GET_LSB(tos))
   10 {
   11 board.w_attacks |= from;
   12 movelist[movenr].from = from;
   13 movelist[movenr].to = to;
   14 movelist[movenr].det.ll = 0;
   15 movelist[movenr].det.ail.piece = ROOK;
   16 movelist[movenr].det.ail.from_nr = from_nr;
   17 movelist[movenr].det.ail.flags |= FROM_NR_COMPUTED;
   18 if (board.b_occ & to) movelist[movenr].det.ail.flags |= NORMAL_CAPTURE;
   19 movenr++;
   20 CLEAR_LSB(tos);
   21 };
   22 CLEAR_LSB(froms);
   23 };
```

For generating rook-moves, the idea of "rotated biboards" (section 1.4) comes into play. But at first, the white rooks are computed and extracted in lines 2-3, and the number of the square where the rook is standing is computed is line 5 and stored in "from_nr" (see previous sections for details). In lines 5 and 6 patterns of the rank and the file on which the rook is standing is saved in "rank_pattern" and "file_pattern" respectively. These patterns are 8-bit variables that are used to index the "rank_moves" and "file_moves"-arrays in line 8, in addition to "from_nr", containing the square where the rook is standing (see section 1.3.3 for details). In line 7, you can see how the idea of accessing the "rotated" representations of the occupied squares works in practice: The desired file-pattern is found in

"`board.occ_l90.byte[l90_to_normal[from_nr] >> 3]`"

Let's look at the individual parts of this expression:

- "board.occ_l90" contains a bitboard of the occupied squares, shifted by 90° to the left
- this bitboard consists of 8 bytes (i.e. 64bits), that can be accessed individually by "board.occ_l90.byte[0]" to "board.occ_l90.byte[7]"
- in order to get the correct byte-number, the "from_nr" is converted to the "l90"-square-nr by accessing the array "l90_to_normal" with index "from_nr" and shifted to the right by 3 bits
- this last shift can also be seen in line 6. When shifting "from_nr" (a number from 0...63) to the right by 3 bits, you will get the number of the byte where the bit corresponding to the "from_nr" resides

Thus, after line 6 and 7, you have the correct patterns stored in "rank_pattern" and "file_pattern". These are used to access the pre-computed "rank_moves" and "file_moves" arrays in line 8, where the bitboard of the possible destination squares for rook-moves ("tos") is computed. The individual moves are put in the movelist in lines 9-23 as decribed above.

2.4.2 Bishops

Here the a code-snippet from the FUSc#-move-generator that generates moves for the white rook:

```
   1  // WHITE BISHOP
   2  froms = board.bishops & from_squares;
   3  while (from = GET_LSB(froms))
   4  {
   5  from_nr = get_LSB_nr(from);
   6  a1h8_pattern = board.occ_a1h8.byte[a1h8_to_normal[from_nr] >> 3];
   7  a8h1_pattern = board.occ_a8h1.byte[a8h1_to_normal[from_nr] >> 3];
   8  tos = (a1h8_moves[from_nr][a1h8_pattern] | a8h1_moves[from_nr][a8h1_pattern])
& to_squares;
   9  while (to = GET_LSB(tos))
10 {
11 board.w_attacks |= from;
12 movelist[movenr].from = from;
13 movelist[movenr].to = to;
14 movelist[movenr].det.ll = 0;
15 movelist[movenr].det.ail.piece = BISHOP;
16 movelist[movenr].det.ail.from_nr = from_nr;
17 movelist[movenr].det.ail.flags |= FROM_NR_COMPUTED;
18 if (board.b_occ & to) movelist[movenr].det.ail.flags |= NORMAL_CAPTURE;
19 movenr++;
20 CLEAR_LSB(tos);
21 };
22 CLEAR_LSB(froms);
23 };
```

The move generation for bishops is quite similar to the move generation for rooks. In line 6 and 7 the needed patterns (this times for the two diagonal directions) are computed, whereas the arrays with the pre-computed moves are accessed in line 8. Note that for bishop moves, the corresponding rotated bitboards are called "board.occ_a1h8" and "board.occ_a8h1". For details on how these rotated bitboards look like, please have a look at the Appendix (5). Again, the individual moves are put in the movelist in lines 9-23, as described earlier.

2.4.3 Queens

The queens moves are generated in the same way as the rook and bishop moves. Basically, the idea is to generate bitboards for all rank/file/diagonal moves from the square the queen is standing on, and "ORing" all of those bitboards in order to get a bitboard with all the queen moves. Again, the individual moves are then put in the movelist, as described earlier.

3 Verifying the move-generator in FUSc#

Constructing a basic move-generator is not too hard, since the basic rules for piece-movements in chess are manageable both in number and complexity. However, when also considering special moves like castling, promotion and en-passent and the huge number of possible chess positions there are some really tricky cases to handle - and the question arises how to make sure that the move-generator of one's chess program works 100% correct, even in awkward and seldom ocurring yet possible positions. "Manually" checking the move-lists of the program is possible for only a very limited number of positions - nevertheless it should of course be done in the process of developing a chess program, although it can always only be a first step. A more advanced method to verify the move-generator of a chess engine have been developed is to use the command "perft".

3.1 The "perft"-idea

The basic idea is to implement a "perft"-command to the chess engine which will construct a minmax-tree untill a fixed depth and count all the generated nodes. This number can be compared to the number of nodes generated by the "perft"-command of other chess engines, and there exist Web-Sites with both a collection of chess positions and the corresponding correct "perft"-numbers for several depths (see [10]). Of course, special attention should be given to positions involving "special moves" like castling, en-passent and promotion. One important point is that the search conducted by the "perft"-command must construct a plain minmax-tree without alpha-beta, transpositions tables, quiescence search, search extensions or any forward pruning techniques like null-moves.

3.2 Test positions

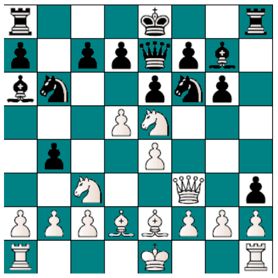
3.2.1 The start postion

The correct results for the “perft”-command at the start position are given in the following table. It is clear that for depth 1 (i.e. “move generation for white and counting the nodes”) the result is 20, as there are 20 legalmoves for white in the start position in chess:

Depth	Perft(Depth)
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324
7	3,195,901,860
8	84,998,978,956
9	2,439,530,234,167
10	69,352,859,712,417

3.2.2 A middlegame position

The following position involves castling, en-passent and promotion (at least in higher depths) for both sides.
The FEN-Code is r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq -

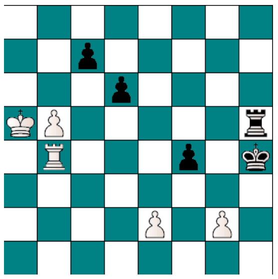


The correct results are:

Depth	Perft(Depth)
1	48
2	2039
3	97,862
4	4,085,603
5	193,690,690
6	8,031,647,685

3.2.3 An endgame Position

Here is an endgame-poistion with FEN-code 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -



The correct results are:

Depth	Perft(Depth)
1	14
2	191
3	2,812
4	43,238
5	674,624
6	11,030,083
7	178,633,661

3.3 Results of Crafty and FUSc#

In order to test the move-generator of FUSc#, the three positions described above were loaded into the program and the "perft"-command was executed. In order to re-check the results, the experiment was also done with Crafty (version 1919p3, see [9]). See the Appendix (5.2) for the detailed output of both programs in the three positions.

4 Conclusions and further research

In this paper, the ideas behind the rotated bitboard board representations were explained and illustrated at the chess program Fusch#, which uses this technique in order to construct an efficient the move-generator. When we introduced this concept to our chess program, we observed a huge speedup compared to the old version of FUSc#, which used the array-representation. However, when it comes to speed, other chess programs like "Crafty" perform much better in terms of "nodes per second", which is closely related to the speed of the move-generator (this can also be seen at the execution time of the "perft"-command, which is also much faster in Crafty than in FUSc#). FUSc# is written in C# and runs on Microsoft Framework .NET, which means that the source-code of FUSc# is not compiled into machine language directly, but into the "Microsoft Intermediate Language" (MSIL), which is translated into machine language by the JIT-compiler ("Just in Time") of the .NET-Framework **at execution time**. Crafty, on the contrary, was written in C, is compiled directly into machine language, and uses compilers that makes intensive use of optimizations **at compiling time** (like gcc, see). That's why a big part of the difference in performance can be explained by the use of different programming frameworks - .NET was not made for low-level high-performance applications in the first place, but for distributed computing, web services etc. Additionally, FUSc# was in the past mainly a research project not with the aim of fine-tuning the move-generator or the search function of chess programs, but to experiment with new ideas in chess programming like machine learning, neuronal networks (see [7]). That's why it is understandable that it can not compete with professional programs that were developed and tuned for many years by professional programmers and/or chess players. Nevertheless, the performance and chess skill of FUSc# have improved steadily over the last three years although the development of FUSc# was done by students in their free time, the team was changing often etc. It is has successfully playing at the FUSc#-servers for move than one year now, and has all features of modern uci-engines as well as some interesting additions like a self-learning opening book.

An interesting open question is if it is possible to construct a high-perfomance chess playing programm for the .NET-Framework. To our knowledge, FUSc# is the only mature chess engine written for .NET. In general, there are some possibilities in the .NET-Framework for performance tuning that were not used in FUSc# until now, like the utilization of “inline-assembly” or the usage of “.NET-compilers”. An interesting perspective arises with the arrival of 64bit processors (like AMD64-chips, that are relatively cheaply available), because first experiments show that the .NET-Framework and FUSc# heavily profit from the 64bit enviroment (the “nps” were roughly doubled without any source code changes!). Even more promising could be port of FUSc# to the new “64bit-Dualcore”-CPUs, although this step would need a rewrite of major parts of the program (e.g. conducting a parallel the alpha-beta-search). Nevertheless, this could be an interesting perspective, as there are many multiprocessor systems based on AMD64-processors (e.g. with AMD Opteron) available, that could form a base for future running enviroments of a “DeepFUSc#”-program.

5 Appendix

5.1 Rotated bitboards in detail

This section should illustrate how the chess-board looks like in the “rotated bitboards”. After the “normal” bitboard, the flipped bitboard (rotated to the left by 90°) as well as the two bitboards needed for move generation in direction of the two diagonals (the “a1h8” and the “a8h1”-bitboard) will be shown. For more information, please have a look at [2].

5.1.1 The normal bitboard

This is the “normal” bitboard, as used in many places in the program:

```
a8 b8 c8 d8 e8 f8 g8 h8
a7 b7 c7 d7 e7 f7 g7 h7
a6 b6 c6 d6 e6 f6 g6 h6
a5 b5 c5 d5 e5 f5 g5 h5
a4 b4 c4 d4 e4 f4 g4 h4
a3 b3 c3 d3 e3 f3 g3 h3
a2 b2 c2 d2 e2 f2 g2 h2
 a1 b1 c1 d1 e1 f1 g1 h1
```

5.1.2 The flipped bitboard (“l90”)

The “flipped” bitboard is stored in “board.occ_l90” and used to generate moves along files for rooks and queens:

```
a8 a7 a6 a5 a4 a3 a2 a1
b8 b7 b6 b5 b4 b3 b2 b1
c8 c7 c6 c5 c4 c3 c2 c1
d8 d7 d6 d5 d4 d3 d2 d1
e8 e7 e6 e5 e4 e3 e2 e1
f8 f7 f6 f5 f4 f3 f2 f1
g8 g7 g6 g5 g4 g3 g2 g1
h8 h7 h6 h5 h4 h3 h2 h1
```

5.1.3 The a1h8 bitboard

The a1h8 bitboard is stored in "board.occ_a1h8" and used to generate diagonal moves in direction of the "a1h8-diagonal" for bishops and queens. Note that in this "compressed" representation, it must be assured that a piece can not "jump over the edge" of the chessboard and re-enter it on the other side because this would allow illegal moves. This must be taken care of during the initialisation of the bitboards, where all the legal diagonal moves in the direction of the a1h8-diagonal are encoded into the "a1h8_moves"-array. The edge of the board is marked with the symbol "|" in the following figure:

```
a8 | b1 c2 d3 e4 f5 g6 h7
a7 b8 | c1 d2 e3 f4 g5 h6
a6 b7 c8 | d1 e2 f3 g4 h5
a5 b6 c7 d8 | e1 f2 g3 h4
a4 b5 c6 d7 e8 | f1 g2 h3
a3 b4 c5 d6 e7 f8 | g1 h2
a2 b3 c4 d5 e6 f7 g8 | h1
 a1 b2 c3 d4 e5 f6 g7  h8
```

5.1.4 The a8h1 bitboard

The a1h8 bitboard is stored in "board.occ_a1h8" and used to generate diagonal moves in direction of the "a1h8-diagonal" for bishops and queens. The edge of the board is again marked with the symbol "|" in the following figure (see above):

```
a8 b7 c6 d5 e4 f3 g2   h1
a7 b6 c5 d4 e3 f2 g1 | h8
a6 b5 c4 d3 e2 f1 | g8 h7
a5 b4 c3 d2 e1 | f8 g7 h6
a4 b3 c2 d1 | e8 f7 g6 h5
a3 b2 c1 | d8 e7 f6 g5 h4
a2 b1 | c8 d7 e6 f5 g4 h3
 a1 | b8 c7 d6 e5 f4 g3 h2
```

5.2 "perft"-output for FUSc# and Crafty

As a proove for the correctness of the FUSc# move generator, the position described in section 3.2 are loaded into FUSc# and Crafty. Then the "perft"-command is executed. All the computed numbers turn out to be correct for both FUSc# and Crafty! For reference, you find the original output in the following two sections.

5.2.1 FUSc#

In FUSc#, the "perft"-command is implemented as "debug counodes", in order to be consistent with the other debugging commands (that can be obtained by entering "debug help" at the command prompt). Here is the output up to depth 5:

```
debug countnodes 1
Minmax-Suche bis Tiefe 1
Besuchte Knoten:  20
debug countnodes 2
```

```
Minmax-Suche bis Tiefe 2
Besuchte Knoten:  400
debug countnodes 3
Minmax-Suche bis Tiefe 3
Besuchte Knoten:  8902
debug countnodes 4
Minmax-Suche bis Tiefe 4
Besuchte Knoten:  197281
debug countnodes 5
Minmax-Suche bis Tiefe 5
Besuchte Knoten:  4865609
```

Now the middlegame-position:

```
position fen r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq -
debug countnodes 1
Minmax-Suche bis Tiefe 1
Besuchte Knoten:  48
debug countnodes 2
Minmax-Suche bis Tiefe 2
Besuchte Knoten:  2039
debug countnodes 3
Minmax-Suche bis Tiefe 3
Besuchte Knoten:  97862
debug countnodes 4
Minmax-Suche bis Tiefe 4
Besuchte Knoten:  4085603
```

And now the endgame-position:

```
position fen 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -
debug countnodes 1
Minmax-Suche bis Tiefe 1
Besuchte Knoten:  14
debug countnodes 2
Minmax-Suche bis Tiefe 2
Besuchte Knoten:  191
debug countnodes 3
Minmax-Suche bis Tiefe 3
Besuchte Knoten:  2812
debug countnodes 4
Minmax-Suche bis Tiefe 4
Besuchte Knoten:  43238
debug countnodes 5
Minmax-Suche bis Tiefe 5
Besuchte Knoten:  674624
debug countnodes 6
Minmax-Suche bis Tiefe 6
Besuchte Knoten:  11030083
```

5.2.2 Crafty

Here is the output of Crafty in the start-position:

```
White(1):  perft 1
total moves=20 time=0.00
White(1):  perft 2
total moves=400 time=0.00
White(1):  perft 3
total moves=8902 time=0.00
White(1):  perft 4
total moves=197281 time=0.26
White(1):  perft 5
total moves=4865609 time=6.66
White(1):  perft 6
total moves=119060324 time=283.79
```

Now the middlegame-position:

```
White(1): setboard r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w
KQkq -
White(1): perft 1
total moves=48 time=0.00
White(1): perft 2
total moves=2039 time=0.00
White(1): perft 3
total moves=97862 time=0.10
White(1): perft 4
total moves=4085603 time=4.52
```

And now the endgame-position:

```
White(1): setboard 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -
White(1): perft 1
total moves=14 time=0.00
White(1): perft 2
total moves=191 time=0.00
White(1): perft 3
total moves=2812 time=0.00
White(1): perft 4
total moves=43238 time=0.04
White(1): perft 5
total moves=674624 time=0.86
White(1): perft 6
total moves=11030083 time=22.34
```

References

[1] Heinz, Ernst A., *Scalable search in computer chess,* Vieweg, 2000

[2] Ernst A.Heinz: *How DarkThought plays chess*, http://supertech.lcs.mit.edu/~heinz/dt/node2.html

[3] Frey, P.W. (editor), *Chess skill in man and machine*, Springer, 2nd edition 1983

[4] Plaat, Aske: RESEARCH, RE:SEARCH & RE-SEARCH, Tinbergen Institute, 1996

[5] FUSc#-Homepage: http://www.fuschmotor.de.vu

[6] Download of DarkFUSc# and FUSc#: http://page.mi.fu-berlin.de/~fusch/download/download_de.html

[7] Block, Marco, *Verwendung von Temporale-Differenz-Methoden im Schachmotor FUSc#,* Diplomarbeit, Berlin, 2004

[8] Dill, Sebastian, *Transpositionstabellen*, Seminararbeit, Berlin, 2005

[9] Crafty-homepage http://www.cis.uab.edu/info/faculty/hyatt/hyatt.html

[10] Homepage of Peter McKenzie on Computer chess: http://homepages.caverock.net.nz/~peter/perft.htm

[11] Homepage Microsoft .NET: http://www.microsoft.com/net/default.mspx

[12] Homepage of gcc: http://gcc.gnu.org/