USING BITBOARDS FOR MOVE GENERATION IN SHOGI

Reijer Grimbergen[1]

Yamagata, Japan

ABSTRACT

In this paper it will be explained how to use bitboards for move generation in shogi. In chess, bitboards have been used in most strong programs because of the easy representation of a chess board by a single 64-bit integer. For shogi, a less efficient representation has to be used because a shogi board has 81 squares instead of 64. A representation with an array of three integers is proposed, where each integer represents 27 squares of the board. This representation is then used for move generation in a similar way to the methods used in chess, for example by using rotated bitboards for generating the moves of the sliding pieces Rook and Lance. For generating the moves of the Bishop, rotated bitmaps are not completely satisfactory and a different method using the alignment of diagonals is proposed for move generation. A comparison of the move-generation speed between using bitboards and the more common method of using attack tables showed that even without using hardware dependent optimizations the move-generation speed of the shogi program SPEAR using bitboards was more than 40% faster than the move generation using attack tables.

1. INTRODUCTION

Bitboards are a binary representation of knowledge for all squares on the board. The presence of certain information for a square is represented by setting the bit for this square on the board to 1, while the absence is represented by 0. Bitboards have been used extensively in chess programs (Adelson-Velskiy *et al.*, 1970; Slate and Atkin, 1977; Cracraft, 1984; Wendroff, 1985; Hyatt, Gower, and Nelson, 1990; Heinz, 1997; Hyatt, 1999b). The reason for this is that the size of the board ($8 \times 8 = 64$ squares) makes it possible to fit one bitboard into a single 64-bit integer. Therefore, the representation of bitboards is very efficient (no unused memory) and representing new knowledge by combining the information from different bitboards is very fast because logical operations like AND, OR, NOT and XOR can be used directly.

Until recently, the bitboard approach has not been given much attention in games other than chess. As a shogi programmer, I envied the computer-chess community for having a game with a 64-square board that makes such an elegant representation possible. The shogi board has 81 squares (9×9) and a bitboard representation seems to be too inefficient to be useful. In my shogi program SPEAR (and most shogi programmers use a similar approach), I have been using simple attack tables to keep information about where pieces can move to and how many pieces can move to a certain square. These tables are updated incrementally and a stack is used to restore the information in the attack tables in case of undoing a move. Quite a complicated piece of code and an area of the program that I rather not touch even if there are some obvious optimizations that can be implemented.

Recently, a combination of events sparked my interest in bitboards for shogi. First, the difficult preparation of my shogi program for last year's World Computer Shogi Championships and the rather disappointing result in the tournament showed that there were some fundamental flaws in the program. Second, the tournament was won by first-year entry BONANZA, which was rumored to use bitboards. Finally, a discussion on the computer shogi bulletin board of Hiroshi Yamashita (2005) (the programmer of former world champion YSS and runner-up in last year's championship) about using bitboards in shogi was brought to my attention. Yamashita had tried bitboards in shogi and reported a 20% increase in speed, which he found disappointing. My feelings were

[1]Department of Informatics, Yamagata University, Yamagata, Japan. Email: grim@yz.yamagata-u.ac.jp

completely different: if bitboards are good enough to improve YSS, they are definitely good enough to improve my program. Furthermore, further study showed that Yamashita's bitboard implementation was far from optimal.

I spent the summer rebuilding my shogi program from scratch with bitboards instead of attack tables and my findings are that bitboards are a very promising representation in shogi as well, despite the obvious inefficiencies that are the result of having a board with 81 squares. In this paper I will first explain in Section 2 how bitboards have been used in chess. In Section 3 I will then give details about using bitboards in shogi. The experiments in Section 4 show that by using bitboards, the move-generation speed can be improved by more than 40% when compared to attack tables. This is only a first step into the use of bitboards in shogi and in Section 5 a number of ideas for further improvements are given.

2. BITBOARDS IN CHESS

In chess, bitboards were first used in the famous KAISSA program (Adelson-Velskiy *et al.*, 1970) in the late 1960s. The first detailed description of how to use bitboards for generating moves was given by Slate and Atkin (1977). This was done by incrementally updating sets of bitboards that contain the information of where a piece on a square can move to (`attacks_from[BOARDSIZE]`) and information about which pieces can move to a square (`attacks_to[BOARDSIZE]`). With this information, it is easy to generate the moves for non-sliding pieces by taking the AND of the corresponding `attacks_from` bitboard with the bitboard having 0s for the friendly pieces and looping through the set bits.

```
unsigned_64 target = ~FriendlyPieces;
from = PieceSQ;     // PieceSQ is the square of the non-sliding piece
unsigned_64 piece_moves = attacks_to[from] & target;
while(piece_moves) {
   // Find the next 1 in the bitboard
   to = FirstOne(piece_moves);
   // Generate the move with the available information
   generate_move(Piece, from, to);
   // Clear the 1 in the piece moves bitboard
   Clear(to, piece_moves);
}
```

The problem is with the sliding pieces (Queen, Rook, and Bishop in chess), because the square to where these pieces can move, depends on which pieces are blocking the paths of the sliding pieces. This information is only available during the game and early attempts had bitboards for the rays of sliding pieces (i.e. a 1 for every square the piece could move to if blocking was ignored) and focused on how to strip off efficiently the bits beyond a blocking piece. An explanation of this rather complicated method is given by Hyatt (1999b).

An important improvement of this method was given by Wendroff (1985), who pre-calculated the attacks for all possible blocking patterns for all possible squares, so that only a blocking pattern had to be calculated during the game (which is just taking the AND of the non-occupied squares and the rays of the piece). For example, the horizontal attack patterns for all squares for a Rook in chess can be pre-calculated and stored in the bitboard array `horizontal_attacks[64][256]`. Generating the horizontal moves for a Rook is then straightforward.

```
from = RookSquare;
// Isolate the bits on Rook rank that represent occupied squares
int blocking = GetRankBits(FriendlyPieces | OpponentPieces, from);
// This pattern can now be used to find the pieces that a Rook can move to
unsigned_64 piece_moves = horizontal_attacks[from][blocking];
while(piece_moves) {
   to = FirstOne(piece_moves);
   generate_move(ROOK, from, to);
   Clear(to, piece_moves);
}
```

The size of the array `horizontal_attacks[64][256]` is is $8 \times 64 \times 256$ = 131 KB, which is quite a big chunk of memory. In an addition to his ICCA paper, Hyatt (1999a) explained that the occupation of the end squares of a rank is irrelevant (the attack will be the same whether these squares are occupied or not). Therefore, the bitboard array can be reduced to `horizontal_attacks[64][64]` which is only 32 KB. The blocking

pattern calculation becomes slightly more complicated in the sense that the end bits of the rank pattern have to be stripped.

This method only works well if the block pattern bits are adjacent, which is not the case for vertical attacks and for diagonal attacks. A solution to this problem that was given by Heinz (1997) is to force the bits in adjacent positions by rotating the board. For vertical attacks, a 90 degree rotation will ensure that ranks become files and files become ranks. This bitboard must be updated at every move, which is a slightly inefficient, but if this information is available, it is again possible to pre-calculate all the vertical attack patterns and use these together with the rotated occupation information to get the vertical squares a Rook can move to.

```
from = RookSquare;
// Isolate the bits on Rook rank that represent occupied squares in rotated form
int blocking = GetRankBits(FriendlyPiecesRotated | OpponentPiecesRotated, from);
// Strip the end bits
blocking = (blocking & 127) >> 1;
// This pattern can now be used to find the pieces that a Rook can move to
unsigned_64 piece_moves = vertical_attacks[from][blocking];
while(piece_moves) {
   to = FirstOne(piece_moves);
   generate_move(ROOK, from, to);
   Clear(to, piece_moves);
}
```

For diagonals the situation is a little bit more complex, because the number of diagonals is larger than the number of ranks and the lengths of the diagonals vary from 1 to 8. Heinz (1997) solved this problem by aligning the diagonals such that the squares of each diagonal are represented by adjacent bits. Hyatt (1999b) stayed closer to the rotation idea by forcing the diagonal squares in adjacent bits by using board rotations of 45 degrees. There is not much difference between these two methods in chess, but it will be explained later that in shogi the alignment of diagonal bits is important.

There are some additional problems with using bitboards in chess, like dealing with en-passant captures and castling, but these will be omitted because they do not play a role in shogi.

3. BITBOARDS IN SHOGI

In a sense, shogi is a more regular game than chess. As mentioned, there are no en-passant captures and special castle moves. All pieces capture in the direction they move. Also, even though the piece movement of most pieces is different from chess, there are only three sliding pieces: Rook, Bishop, and Lance. The movement of the Rook and Bishop is identical to the Rook and Bishop in chess, which leaves only the Lance as a special case. Another difference between shogi and chess is that in shogi almost all pieces can promote. However, most pieces promote to the same piece (a Gold general) and there are no promotions to different sliding pieces. Therefore, to use bitboards in shogi the main problem lies in dealing with the larger board.

3.1 Basic Bitboards

The chessboard has 64 squares, which means that the information about a position fits into a single 64 bit integer. In shogi, this information has to have 81 bits, which will not fit into a single integer. Like Yamashita (2005), I use three integers for representing the 81 bits of the board. Yamashita defined a structure of three integers, but as Hoki (2005) pointed out, it is better to define an array of length 3 (why this is better will become clear later).

```
typedef struct {
   unsigned int bb[3];
} BITBOARD;
```

Because of the different board sizes of chess and shogi, for each bitboard there will 15 bits that are obsolete (3×32 bits-81 bits = 15 bits). Therefore, 16% of the memory allocated to bitboards will not be used. This cannot be helped, but it is also the reason why three 32 bit integers were used for the representation instead of two 64 bit integers (in which case 37% of the bitboard memory would not be used).

9	8	7	6	5	4	3	2	1	
0	1	2	3	4	5	6	7	8	a
9	10	11	12	13	14	15	16	17	b
18	19	20	21	22	23	24	25	26	c
27	28	29	30	31	32	33	34	35	d
36	37	38	39	40	41	42	43	44	e
45	46	47	48	49	50	51	52	53	f
54	55	56	57	58	59	60	61	62	g
63	64	65	66	67	68	69	70	71	h
72	73	74	75	76	77	78	79	80	i

	9	8	7	6	5	4	3	2	1	
	26	25	24	23	22	21	20	19	18	a
bb[0]	17	16	15	14	13	12	11	10	9	b
	8	7	6	5	4	3	2	1	0	c
	26	25	24	23	22	21	20	19	18	d
bb[1]	17	16	15	14	13	12	11	10	9	e
	8	7	6	5	4	3	2	1	0	f
	26	25	24	23	22	21	20	19	18	g
bb[2]	17	16	15	14	13	12	11	10	9	h
	8	7	6	5	4	3	2	1	0	i

Figure 1: Square numbers of a shogiboard (left) and the correspondence between shogiboard squares and bitboard integer bits (right).

Each of the three integers represents one third of the board. In my representation, the first 27 bits of the first integer represent squares 1c to 9a. The first 27 bits of the second integer represent the squares 1f to 9d. Finally, the first 27 bits of the third integer represent the squares 1i to 9g (see Figure 1). Splitting the board up in this way is different from Yamashita's presentation, but an important optimization. The squares 1c to 9a are the squares on which the black pieces can promote, while the squares 1i to 9g are the squares where the white pieces can promote[2]. With this representation it is often possible to access only one bitboard integer when generating promotions instead of having to access all three bitboard integers.

To manipulate bitboards in chess, straightforward logical operations can be used. However, when a bitboard consists of three different integers, this is no longer possible. Logical operations like taking the AND of two bitboards have to be defined as follows:

```
void AndBB(BITBOARD *to, BITBOARD *from1, BITBOARD *from2) {
   to->bb[0] = from1->bb[0] & from2->bb[0];
   to->bb[1] = from1->bb[1] & from2->bb[1];
   to->bb[2] = from1->bb[2] & from2->bb[2];
}
```

In my implementation, I use similar operations for `OR`, `XOR` and `NOT`. Furthermore, I use the basic operations `ClearBB` (setting all bits to 0), `SetBB` (setting all bits to 1), and `CopyBB` (copying a bitboard).

With the basic bitboard data structure defined like this, it is now possible to represent the knowledge needed in a shogi program in a similar way to the bitboards in chess. For example, useful information is to know which squares are occupied. For the starting position of shogi, the bitboard `Occupied` has 1s for the squares with a black or white piece and 0s for squares without pieces (Figure 2).

In the `BITBOARD` data structure this would be:

```
bb[0]: 00000111111111010000010111111111
bb[1]: 00000000000000000000000000000000
bb[2]: 00000111111111010000010111111111
```

My implementation has bitboards for the occupied squares (`Occupied`), the black pieces (`BlackPieces`), the white pieces (`WhitePieces`), and for each black and white piece separately. These bitboards for position information must be updated each time the position changes, i.e., each time either player makes a move.

[2] In shogi, the black player is the player moving up the board, while the white player moves down the board.

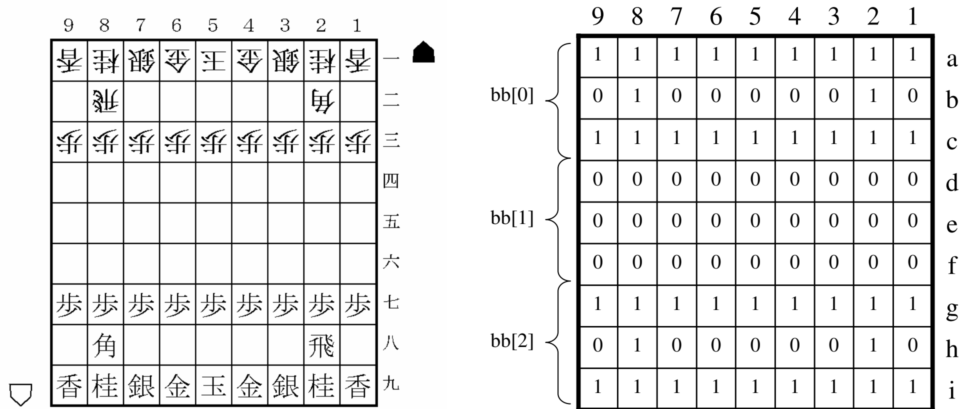

Figure 2: Bit settings for the starting position in shogi.

3.2 Attack Bitboards

Shogi has the following pieces:

- **Non-sliding pieces**: King, Gold general, Silver general, Knight, Pawn,
- **Sliding pieces**: Rook, Bishop, Lance.

The Silver general, Knight, Pawn, and Lance promote to Gold general, which is also a non-sliding piece. The Bishop and Rook promote to Dragon and Horse. The Dragon is a Rook with the single diagonal squares added to the movement, while the Horse is a Bishop with the single horizontal and vertical squares added. Therefore, Dragon and Horse can be handled in the same way as Rook and Bishop.

The non-sliding pieces are handled exactly the same way as in chess, only the initialization of the pre-calculated bitboards has to be different based on the different movements of the pieces.

For the Rook, using bitboards is also very similar to the method explained by Hyatt (1999b). The only difference is that ranks and files are one square longer, so the pre-calculated bitboards are larger. Let us look at the calculation method for horizontal attacks as an example.

First, the definition of the pre-calculated bitboards:

```
BITBOARD RankAttacks[81][128];
```

These bitboards are filled with all the possible block patterns at the start of the program.

The horizontal attacks for a Rook on a square are then calculated as follows.

```
void CalcRankAtt(BITBOARD *bb, int from) {
   // Get the block pattern from the position bitboard
   int blocking = GetRankBits(&Occupied, from);
   // Strip the bits for the edge squares
   blocking = (blocking & CLEAR9MASK) >> 1;
   // Get the horizontal attacks from the pre-calculated bitboards
   CopyBB(bb, &RankAttacks[from][blocking]);
}

int GetRankBits(BITBOARD *bb, int sq) {
   // Take the rank bits from bitboard integer 0, 1, or 2
   return (bb->bb[sq_to_bb_index[sq]] >> rank_shift_no[sq]) & ALLONES_9;
}
```

```
int sq_to_bb_index[] = {
   0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1,
   2, 2, 2, 2, 2, 2, 2, 2, 2,
   2, 2, 2, 2, 2, 2, 2, 2, 2,
   2, 2, 2, 2, 2, 2, 2, 2, 2,
};

int rank_shift_no[] = {
  18, 18, 18, 18, 18, 18, 18, 18, 18,
   9,  9,  9,  9,  9,  9,  9,  9,  9,
   0,  0,  0,  0,  0,  0,  0,  0,  0,
  18, 18, 18, 18, 18, 18, 18, 18, 18,
   9,  9,  9,  9,  9,  9,  9,  9,  9,
   0,  0,  0,  0,  0,  0,  0,  0,  0,
  18, 18, 18, 18, 18, 18, 18, 18, 18,
   9,  9,  9,  9,  9,  9,  9,  9,  9,
   0,  0,  0,  0,  0,  0,  0,  0,  0,
};
```

It is now clear why having an array of three integers is a better representation than three different integers like in Yamashita's original implementation. By using the array `sq_to_bb_index` to access the bitboard integer that is needed, no if-statements are required.

For vertical attacks, the new bitboard `Occupied_rot90` is needed to represent the occupied squares in case of rotation, like in chess. With this bitboard and the block patterns in `FileAttacks[81][128]`, the file attack code is straightforward:

```
void CalcFileAtt(BITBOARD *bb, int from) {
   // Get the block pattern, rotating the source square 90 degrees
   int blocking = GetRankBits(&Occupied_rot90, init_rot90[from]);
   // Strip the bits for the edge squares
   blocking = (blocking & CLEAR9MASK) >> 1;
   // Get the vertical attacks
   CopyBB(bb, &FileAttacks[from][blocking]);
}

// Translate square to 90 degree rotation square
int init_rot90[] = {
   72, 63, 54, 45, 36, 27, 18,  9, 0,
   73, 64, 55, 46, 37, 28, 19, 10, 1,
   74, 65, 56, 47, 38, 29, 20, 11, 2,
   75, 66, 57, 48, 39, 30, 21, 12, 3,
   76, 67, 58, 49, 40, 31, 22, 13, 4,
   77, 68, 59, 50, 41, 32, 23, 14, 5,
   78, 69, 60, 51, 42, 33, 24, 15, 6,
   79, 70, 61, 52, 43, 34, 25, 16, 7,
   80, 71, 62, 53, 44, 35, 26, 17, 8,
};
```

This solution was given by Hyatt (1999b) for chess and it can be used in shogi as well. In chess it is only used for Rook attacks, but in shogi it can be used for both Rooks and Lances. The difference for a Lance is that a Lance can only move vertically and cannot move back. Therefore, two additional blocking pattern data structures, `BlackLanceAttacks[81][128]` and `WhiteLanceAttacks[81][128]` are needed to translate the possible blocking patterns to the squares the Lance is attacking.

0	9	1	18	10	2	27	19	11
3	36	28	20	12	4	45	37	29
21	13	5	54	46	38	30	22	14
6	63	55	47	39	31	23	15	7
72	64	56	48	40	32	24	16	8
73	65	57	49	41	33	25	17	74
66	58	50	42	34	26	75	67	59
51	43	35	76	68	60	52	44	77
69	61	53	78	70	62	79	71	80

8	7	17	6	16	26	5	15	25
35	4	14	24	34	44	3	13	23
33	43	53	2	12	22	32	42	52
62	1	11	21	31	41	51	61	71
0	10	20	30	40	50	60	70	80
9	19	29	39	49	59	69	79	18
28	38	48	58	68	78	27	37	47
57	67	77	36	46	56	66	76	45
55	65	75	54	64	74	63	73	72

Figure 3: Bit assignment for 45 degrees clockwise rotation (left) and 45 degrees anti-clockwise rotation (right).

```
void CalcBlackLanceAtt(BITBOARD *bb, int from) {
   int blocking = GetRankBits(&Occupied_rot90, init_rot90[from]);
   blocking = (blocking & CLEAR9MASK) >> 1;
   CopyBB(bb, &BlackLanceAttacks[from][blocking]);
}
```

In the case of diagonals, there is a problem with the approaches taken by Hyatt (1999b) and Heinz (1997). For example, when the diagonals are put in adjacent bits using the 45 degree rotations, the bits of the bitboard are assigned as in Figure 3.

In Figure 3 it can be seen that there are diagonals overflowing into the next integer of the bitboard. For clockwise rotation, square 6 is part of the diagonal 54 46 38 30 22 14 6 (9g to 3a) and square 74 is part of the diagonal 74 66 58 50 42 34 26 (7i to 1c). Similarly, for anti-clockwise rotation, the square 62 is part of the diagonal 2 12 22 32 42 52 62 (7a to 1g) and square 18 is part of the diagonal 18 28 38 48 58 68 78 (9c to 3i). This requires an extra check when translating diagonals to adjacent bits, which is inefficient[3].

This problem can be solved by fitting the diagonals in a less natural way, but such that no diagonal overflows into the next integer of the bitboard. The translation I used for the north-west diagonals is given in Figure 4. The north-east diagonals have been fitted in a similar way.

The pre-calculated block pattern bitboards are stored in the following bitboard arrays:

```
BITBOARD DiagonalAttacks_nw[81][128];
BITBOARD DiagonalAttacks_ne[81][128];
```

The calculation procedure for finding the squares that a Bishop is attacking now becomes:

```
void BishopAttacks(BITBOARD *bb, int from) {
   BITBOARD nw;
   int blocking = GetDiagonalBits(&BothPieces_nw, init_nw[from]);
   blocking = (blocking & CLEAR9MASK) >> 1;
   CopyBB(nw, &DiagonalAttacks_nw[from][blocking]);
   blocking = GetDiagonalBits(&BothPieces_ne, init_ne[from]);
   blocking = (blocking & CLEAR9MASK) >> 1;
   OrBB(bb, nw, &DiagonalAttacks_ne[from][blocking]);
}
```

[3] To be exact, square 74 (18 in the anti-clockwise version) is not a problem, because this is an edge square at the top of the diagonal and will be stripped anyway. Therefore, this overflow into the next integer of the bitboard (from `bb[1]` to `bb[2]`) can be ignored. Square 6 (62 in the anti-clockwise version) is also an edge square, so its contents can be ignored. However, because this time the square is at the bottom of the diagonal (i.e., the most significant bit) there is the problem of fitting the remaining 6 bits of the diagonal to the 7 bit block information. A single left shift is needed while for all the other diagonals a right shift is needed. Therefore, these two diagonals are a special case and a condition has to be added to the function that gets the block information from the rotated boards.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

0	10	20	30	40	50	60	70	80
1	11	21	31	41	51	61	71	9
19	29	39	49	59	69	79	7	17
2	12	22	32	42	52	62	18	28
38	48	58	68	78	3	13	23	33
43	53	27	37	47	57	67	77	8
4	14	24	34	44	36	46	56	66
76	5	15	25	35	45	55	65	75
6	16	26	54	64	74	63	73	72

Figure 4: Bit assignment for north-west diagonals.

```
int GetDiagonalBits(BITBOARD *bb, int sq) {
   return (bb->bb[square_to_bb_dia_index[sq]] >> dia_shift_no[sq]) & dia_mask[sq];
}

int init_nw[] = {
    0,  9, 27, 41, 54, 64, 72, 25, 53,
   17,  1, 10, 28, 42, 55, 65, 73, 26,
   34, 18,  2, 11, 29, 43, 56, 66, 74,
   47, 35, 19,  3, 12, 30, 44, 57, 67,
   59, 48, 36, 20,  4, 13, 31, 45, 58,
   68, 60, 49, 37, 21,  5, 14, 32, 46,
   75, 69, 61, 50, 38, 22,  6, 15, 33,
   78, 76, 70, 62, 51, 39, 23,  7, 16,
   80, 79, 77, 71, 63, 52, 40, 24,  8,
};

int dia_shift_no[] = {
   18, 18, 18, 18, 18, 18, 18, 18, 18,
   10, 10, 10, 10, 10, 10, 10, 10,  2,
    2,  2,  2,  2,  2,  2,  2,  0,  0,
   20, 20, 20, 20, 20, 20, 20, 13, 13,
   13, 13, 13, 13, 13,  7,  7,  7,  7,
    7,  7,  1,  1,  1,  1,  1,  1,  0,
   22, 22, 22, 22, 22, 17, 17, 17, 17,
   17, 13, 13, 13, 13,  9,  9,  9,  9,
    6,  6,  6,  3,  3,  3,  1,  1,  0,
};

int dia_mask[] = {
   511,511,511,511,511,511,511,511,511,
   255,255,255,255,255,255,255,255,255,
   255,255,255,255,255,255,255,  3,  3,
   127,127,127,127,127,127,127,127,127,
   127,127,127,127,127, 63, 63, 63, 63,
    63, 63, 63, 63, 63, 63, 63, 63,  1,
    31, 31, 31, 31, 31, 31, 31, 31, 31,
    31, 15, 15, 15, 15, 15, 15, 15, 15,
     7,  7,  7,  7,  7,  7,  3,  3,  1,
};
```

Version	*3-ply*	*4-ply*
Attack tables	1509s	165648s
Bitboards	905s	97459s
Savings	43%	41%

Table 1: Results of 3-ply and 4-ply minimax searches in 1000 positions.

The `dia_mask` translation is needed because unlike ranks and files, diagonals do not have a fixed length. Therefore, masking is needed to keep only the information that has the precise length of the diagonal on which the particular square is located. The `init_ne` array has been omitted, but is similar to the north-west array.

To generate moves, the only thing that remains is to decide if the piece can promote or not. For this, bitboards can be used that have 1s for the squares inside the black or white promotion zone. The promotion zone is represented by a single integer, so checking if a square is in the promotion zone can be done by checking only `bb[0]` (for black pieces) or `bb[2]` (for white pieces). In the same way, bitboards can be used to check if the promotion of a Lance, Knight or Pawn is obligatory (Lance and Knight on the top two ranks and Pawn on the top rank).

4. EXPERIMENTAL RESULTS

In the literature, no general method for testing and comparing bitboard techniques has been given. In the papers by Heinz (1997) and Hyatt (1999b) there is no data comparing the performance of bitboards against other methods. The reason for this is probably that bitboards influence many parts of a program, making it hard to do exactly the same thing (e.g., move generation, evaluation, static exchange evaluator etc.) with and without bitboards. To compare bitboards and attack tables in shogi, I have focused on move generation, as this is part of every program. This comparison has also been used by Yamashita (2005), whose implementation of bitboards is similar to the one presented in this paper.

Yamashita (2005) compared the performance of bitboards with the performance of attack tables for move generation in mating problems and found that using bitboards makes his mate solver about 20% faster. However, comparing the performance for mating problems is not a good idea because generating defense moves against checks is relatively costly with bitboards. Specifically, interposing pieces between a checking piece and King is expensive. Most chess programs using bitboards like CRAFTY therefore generate moves normally irrespective of checks and delete moves that leave the King in check on the next ply.

Therefore, here a different test has been performed. To make a clean comparison between bitboards and attack tables, the performance of a depth-limited minimax search without quiescence search in which the moves were generated by the attack tables used in the shogi program SPEAR was compared with a program in which the moves were generated using bitboards. The evaluation function only calculated the material balance. The use of depth-limited minimax assures that for both attack tables and bitboards the number of generated moves is exactly the same. Therefore, the generated search trees in both cases were exactly the same size. Time differences between the searches are therefore only caused by the difference between attack tables and bitboards.

In this way, a 3-ply and a 4-ply minimax search was conducted for 1000 different positions taken from professional games. This experiment was run on a 3.0GHz Pentium 4 machine under WindowsXP. The results are summarized in Table 1. The savings over using bitboards are more than 40%.

5. CONCLUSIONS AND FUTURE WORK

In this paper it was explained how bitboards can be used to do move generation in shogi. Even though the implementation of bitboards in shogi is less efficient than in chess because of the different board size, it has been shown that the speed of move generation in shogi can be improved significantly if bitboards are used instead of attack tables. From the result of Table 1 we may conclude that the move generation using bitboards is more than 40% faster than the move generation with attack tables. Therefore, it seems likely that bitboards are a good alternative for attack tables.

However, this is not the final conclusion about bitboards in shogi. One problem with bitboards is that the in-

formation about squares is only binary. If multi-valued information is needed, bitboards are not so effective. In shogi, attack tables are also used in the evaluation function, especially when calculating piece mobility and the danger on squares around the King. If it is important for a good evaluation of king danger to know the balance between the number of attacking and defending pieces, evaluation will be slower with bitboards than with attack tables. World champion BONANZA is currently not evaluating this balance, but only counts the number of pieces that are attacking and defending, something that can be done efficiently with population-count functions.

There are still many ways in which the efficiency of the bitboard representation presented in this paper can be improved. One area of improvement is the use of hardware specific optimizations. For example, the `FirstOne` function to find the first bit in a bitboard is currently a pure software function. Another optimization is the idea of *magic multiplication*, which has recently been discussed on Internet computer-chess forums. This method makes the iterative updating of rotated bitboards unnecessary. A comparison of magic multiplication and bitboards for chess on the TalkChess.com computer chess forum seems to indicate that a further 10% increase in speed can be expected. It should be noted that this percentage does not translate easily to the data given in this paper, because the shogi engine is too simple. Trying bitboards and magic multiplication in a full shogi engine is therefore an important future work.

It will also be interesting to see how fast the bitboards can be made if two 64-bit integers were used instead of three 32-bit integers. The big memory overhead could be a problem, but it would be good to investigate how much faster bitboards can be if memory issues are ignored. Roughly the opposite idea is to find ways for using the obsolete bits in the bitboard representation. For example, representing (a part of) the edge of the board with the obsolete bits could give optimizations that are not possible in chess.

Finally, move generation speed is not the only reason for using bitboards. Other parts of the program may also benefit from using bitboards. For example, in CRAFTY the static exchange evaluator used in quiescence search is implemented efficiently using bitboards and this can be done in shogi as well.

To conclude, an important extra advantage of bitboards is that they make a program much more transparent. The attack table code of my previous program was buried in a dark corner where I would rather not go, while the bitboard code is much easier to change if there are new ideas for optimizations. I therefore warmly recommend the use of bitboards, even if your game of choice does not have a board where the number of squares is a power of two.

6. REFERENCES

Adelson-Velskiy, G., Arlazarov, V., Bitman, A., Zhivotovsky, A., and Uskov, A. (1970). Programming a Computer to Play Chess. *Russian Mathematical Surveys*, Vol. 25, pp. 221–262.

Cracraft, S. (1984). Bitmap Move Generation in Chess. *ICCA Journal*, Vol. 7, No. 3, pp. 146–153.

Heinz, A. (1997). How DARKTHOUGHT Plays Chess. *ICCA Journal*, Vol. 20, No. 3, pp. 166–176.

Hoki, K. (2005). http://524.teacup.com/yss/bbs. Message posted on August 20th 2005. (In Japanese).

Hyatt, R. (1999a). http://www.cis.uab.edu/hyatt/bitmaps.html.

Hyatt, R. (1999b). Rotated Bitmaps, A New Twist on an Old Idea. *ICCA Journal*, Vol. 22, No. 4, pp. 213–222.

Hyatt, R., Gower, A., and Nelson, H. (1990). CRAY BLITZ. *Computers, Chess and Cognition* (eds. T. Marsland and J. Schaeffer), pp. 111–130. Springer Verlag, New York.

Slate, D. and Atkin, L. (1977). Chess 4.5: The Northwestern University Chess Program. *Chess Skill in Man and Machine* (ed. P. Frey), pp. 82–118. Springer Verlag, New York.

Wendroff, B. (1985). Attack Detection and Move Generation on the X-MP/48. *ICCA Journal*, Vol. 8, No. 2, pp. 58–65.

Yamashita, H. (2005). http://524.teacup.com/yss/bbs. Message posted on August 12th 2005. (in Japanese).