Rotated bitmaps, a new twist on an old idea

Abstract

This paper describes some developments related to using "bitmaps" (64 bit integers using one bit for each square of the chess board). In late 1994, after the ACM computer chess event in Cape May, New Jersey, I decided to embark on a complete replacement chess program for Cray Blitz. I was interested in using the bitmap approach mentioned by Slate and Atkin in chess 4.x to determine for myself whether this approach was suitable for chess or not. In developing this new program, the concept of "rotated bitmaps" was developed, and this turned out to be what was needed to make this type of data structure produce reasonable performance.

1. Introduction

Bitmaps have been mentioned many times in computer chess literature. In the middle 1970's, Slate and Atkin developed the idea and described the approach of using twelve 64-bit integers, one for each type of piece on the board (six for white, six for black). It should be noted that the Kaissa team (Donskoy, et. al.) apparently developed this same idea independently of the Northwestern group, and that many other programs have been written using this approach over the past 25 years.

Slate explained that by numbering the bits to correspond to squares of the chess board, they could use a 1 bit to indicate the presence of a particular piece on a specific square, while a 0 bit indicates the absence of that particular piece. So for white pawns, there are at most eight 1 bits set in the white_pawn bitmap, corresponding to the squares the pawns occupy. (In this paper, the squares are numbered from 0 to 63, with 0 corresponding to square A1, 7 corresponding to square H1, 56 corresponding to square A8 and 63 corresponding to square H8. The bits in a 64 bit integer are numbered with bit 0 being the most significant bit and bit 63 being the least significant bit.)

In addition to storing the basic chess position in twelve of these 64 bit words, Slate and Atkin went on to explain how to construct and incrementally update an additional set of bitmaps that contained two types of information. attacks_from[64] is an array of bitmaps that contain 1 bits for each square attacked by the piece that is on a particular square. For example, attacks_from[E4] gives the squares that the piece on square E4 attacks directly. Likewise, an array of bitmaps attacks_to[64] contains bitmaps that have 1 bits set for any square that is occupied by a piece which attacks the square in question. For example, attacks_to[E4] is a bitmap that contains 1 bits set for any square on the board that contains a piece which directly attacks E4.

These two sets of attack bitmaps are the subject of this paper describing a new and very fast way of dynamically computing the attacks_from[] bitmap without the substantial execution speed penalty associated with the update method described by Slate and Atkin. (It should be noted that Slate/Atkin used an incremental update because the procedure given below is too slow to be used to compute the attacks for sliding pieces "on-the-fly". To make bitmaps work at all, they resorted to incremental update to avoid the high computational cost it incurs.)

There are many other useful characteristics of bitmaps that make their use in a chess program quite beneficial. One of these characteristics is something that might be termed "bit-parallel" operations. For example, in a traditional chess program, if you have a pawn on square E4, and you want to ask the question "is this pawn passed?" you have to check the squares D5, E5, F5, D6, E6, F6, D7, E7 and F7 to be sure that no enemy pawns stand on those squares. That takes nine comparison operations. With a bitmap, you simply pre-compute an array of masks for each square a pawn can stand on, then take this is_passed_white[E4] mask and AND it with the bitmap for black pawns. If you compute this mask correctly, so that it has a 1 bit on each of the above nine squares, that one AND operation can determine the absence or presence of black pawns that prevent this white pawn on E4 from being passed. In effect, we can ask 9 separate questions in parallel, and let one AND operation answer all nine questions at the same time.

There are other benefits as well, but the subject of this paper is rotated bitmaps and how they can be used to produce the attacks_from[64] and attacks_to[64] data as needed, without any incremental updating overhead and without any complex/slow loops.

2. Computing attacks_from[64] using normal bitmaps.

Before going to the rotated bitmap approach, it is necessary to look at the difficulty of computing this attacks_from[] information. For non-sliding pieces it is trivial, of course, because we can pre-compute an array for knight_attacks[64] so that if we have a knight on square F6, knight_attacks[F6] is a bitmap with 1 bits set for each square that a knight on F6 attacks (note that the squares attacked by a knight are independent of the contents of the chess board). The same idea works for the king, and for pawns (with a little work, since pawns capture diagonally but move forward 1 or two squares when not capturing.)

But for sliding pieces, a simple array reference (as above) won't work, because sliding pieces only attack in the directions they slide, and they don't attack in any direction beyond the first piece they encounter in that direction. So, how can we compute this?

First, we are going to use a bitmap called occupied_squares, which is nothing more than a bitmap with a 1 set for any square that is occupied by a piece of either color (ie, this could be thought of as the boolean OR of all twelve separate piece bitmaps although we actually maintain two occupied_squares bitmaps, one for black, one for white, and OR them together to produce this occupied_squares bitmap).

Figure 1 is a sample board position, with occupied squares marked with an X, and empty squares marked with a -.

```
 - - X - - X X -      - - X - - X X -      - - X - - - - -
 X - - X X X X X      X - - X X X X X      - - - X - - - -
 - X X - - - X -      - X X - - - X -      - - - - X - - -
 - X X - - X - -      - X X - - B - -      - - - - - - - -
 X - X - - X - -      X - X - - X - -      - - - - - - - -
 - - X - - X X -      - - X - - X X -      - - - - - - - -
 - X - - X X - X      - X - - X X - X      - - - - - - - -
 X - - X - X X -      X - - X - X X -      - - - - - - - -
    Figure 1.            Figure 2.            Figure 3.
```

Notice that it doesn't matter what is on a square, because a sliding piece stops when it encounters a piece of any kind. If we have a bishop on square F5 (marked by B for clarity in figure 2) then we can see that the bishop directly attacks the following set of squares: {B1,C2,D3,E4,G6,H3,G4,E6,D7}. Once we compute that attack bitmap, if this is to be used as a set of possible white bishop moves in this position, we would want to exclude any moves that would capture our own pieces. By using the white_occupied bitmap mentioned earlier, we can complement this (invert bit by bit) and then AND the result with the attack bitmap, which will clear any bit in the bishops attack bitmap that attacks our own piece since that is not a valid chess move.

So far, so good. But how do we compute the attacked bits without looping down the 4 (or 8 for queen) rays the piece slides along? It turns out to be not so difficult, but it is somewhat inefficient, in that each direction has to be treated with a separate loop iteration.

First, we create a set of masks for the 8 directions a piece can move in. Call these plus1[64], plus7[64], plus8[64], plus9[64], minus1[64], minus7[64], minus8[64], minus9[64]. These bitmaps contain 1 bits on any square that a bishop attacks in the specified direction, starting on square SQ. IE for the current example, with a bishop on F5, in the plus7 direction (which is F5-E6-D7-C8) we get plus7[F5] as shown in figure 3.

That gives us the squares a bishop would attack in the +7 direction, if there were no blocking pieces between the bishop and the edge of the board. But we know that blocking pieces might be present, so we need to find if the bishop is blocked in this direction. To do this, we take the above bitmap, and then AND it with the occupied_squares bitmap. If the bishop is blocked in this direction, we will get a non-zero bitmap which will have a 1 bit set on every square where a piece blocks a bishop sliding down that diagonal toward C8. Since we only care about the first blocking piece (closest to F5) we use the FirstOne() function which returns an index to the first 1-bit set. If the bishop were blocked at C8 and D7, FirstOne() would return the value D7 (51).

Now we have the attacks down this diagonal, and the square where the sliding attack is blocked. All that is left to do is truncate the attacks beyond the blocking piece. And this turns out to be the simplest part. The following snippet of code gives the idea:

```
        diag_attacks=plus7[F5];
        blockers=diag_attacks & occupied_squares;
        blocking_square=FirstOne(blockers);
        diag_attacks^=plus7[blocking_square];
```

That is all that is needed for one ray. To explain the code, diag_attacks is the bitmap described above for the attacks down the +7 direction from the square F5. Blockers becomes a bitmap of any pieces sitting on this particular diagonal in the direction in question. We find the first blocking piece, and then exclusive-OR plus7[blocking_square] with the original diag_attacks which effectively 'cuts off' the attacks beyond that point. (To understand this, if the bishop on F5 is blocked at E6, think about how the plus7[E6] bitmap would look: one bits only on D7 and C8. And when we exclusive-OR those two values, since they both have bits set for D7/C8, those bits (only) get cleared.

We repeat this for all four directions, with the only tricky part being that when we look at the minus directions, we no longer use FirstOne() as that would find the last blocking piece. We switch to using LastOne() instead. Now the code to compute the bishop attacks in all four directions looks like this:

```
        diag_attacks=plus7[F5];
        blockers=diag_attacks & occupied_squares;
        blocking_square=FirstOne(blockers);
        bishop_attacks=diag_attacks^plus7[blocking_square];
        diag_attacks=plus9[F5];
        blockers=diag_attacks & occupied_squares;
        blocking_square=FirstOne(blockers);
        bishop_attacks|=diag_attacks^plus9[blocking_square];
        diag_attacks=minus7[F5];
        blockers=diag_attacks & occupied_squares;
        blocking_square=LastOne(blockers);
        bishop_attacks|=diag_attacks^minus7[blocking_square];
        diag_attacks=minus9[F5];
        blockers=diag_attacks & occupied_squares;
        blocking_square=LastOne(blockers);
        bishop_attacks|=diag_attacks^minus9[blocking_square];
```

This turns into a lot of computation, with the ANDing/ORing and masking necessary, plus using the function FirstOne()/LastOne() to locate the blocking square. This is pretty expensive, but since we are doing lots of work in parallel with the boolean operators, it can still pay off. But there is a better way that eliminates the FirstOne()/LastOne() calls, and which also handles a complete diagonal/rank/file in one step, not the two steps used in this approach.

It should be noted here that the FirstOne()/LastOne() functions are very expensive if they are done as C procedures, because finding the first or last bit set is non-trivial, and this is used many times in computing the above attacks. Fortunately, newer microprocessors have hardware instructions to compute these values (Intel has the BSF/BSR [bit-scan forward, bit-scan reverse] instructions in the X86 architecture) which make these functions extremely fast. Other architectures also offer such instructions.

3. Rotated Bitmaps

When this bit-mapped version of Crafty was started, in September 1994, I formed a seminar group at UAB to study the 64 bit approach and develop new ideas about how to do things efficiently. One of the very first discussions centered on how easy it might be to generate sliding piece moves along a rank.

If you pick any square on a particular rank (horizontal row on the chess board), sliding piece moves are quite obviously easy to generate, because that rank is defined by 8 bits in the occupied_squares bitmap. If you shift this bitmap to position the rank in question in the rightmost 8 bits, and then AND this with 255, we end up with 8 bits that represent the state of the squares on that specific rank.

With that in mind, think about an array of bitmaps called rank_attacks[64][256]. To initialize this array we pick any square, say A1 for this example. Then, we note that this rank can only have one of 256 different states, based on which squares are occupied and which are not. If we compute, up front before starting to play chess, which squares a rook on A1 could move to, given the 256 different configurations of 1 bits on that rank, we can find the attacks along that rank by just accessing the bitmap rank_attacks[A1][contents] where *contents* is the 8 bit rank state described above. All we have to do is compute this set of attack bitmaps for all 64 squares, and for each square, for all of the 256 different rank states (yes, if you think about it, a rank can only have 128 states because we _know_ that a rook is on one of the squares, but it is easier to ignore this compression). It should be mentioned here that while a rank has 8 bits of state, the two end bits (LSB and MSB) don't change anything since they are attacked whether they are empty or occupied. As a result, we therefore strip them off for all of the following calculations to reduce the table size for each by 75%. We will therefore reduce the rank_attacks array to rank_attacks[64][64]. All other such arrays will use the same reduction.

So far, so good, we can get half of the rook attacks with a single memory reference into this rank_attacks array. But what about the file attacks and the two diagonals for bishops? Those bits aren't adjacent in the occupied_squares bitmap which makes this idea impossible to use for anything but the rank.

In discussing this during seminar, we noted that if this were being used in a piece of chess-specific hardware, we would likely define a register that holds the 64 bit occupied_squares value, but that we would create three other "pseudo-registers" that let us access this occupied_squares bitmap in different ways. IE one would rotate the entire occupied_squares 90 degrees so that now bits on a file are adjacent, and we could use this trick to compute the file attacks along with the already-computed rank attacks. And we would also provide two registers that rotate the occupied_squares bitmap left and right 45 degrees, which puts bits on a diagonal into adjacent bits in these rotated bitmaps. Figure 4 illustrates how the squares are numbered in a normal chess board bitmap.

```
A8 B8 C8 D8 E8 F8 G8 H8      H8 H7 H6 H5 H4 H3 H2 H1
A7 B7 C7 D7 E7 F7 G7 H7      G8 G7 G6 G5 G4 G3 G2 G1
A6 B6 C6 D6 E6 F6 G6 H6      F8 F7 F6 F5 F4 F3 F2 F1
A5 B5 C5 D5 E5 F5 G5 H5      E8 E7 E6 E5 E4 E3 E2 E1
A4 B4 C4 D4 E4 F4 G4 H4      D8 D7 D6 D5 D4 D3 D2 D1
A3 B3 C3 D3 E3 F3 G3 H3      C8 C7 C6 C5 C4 C3 C2 C1
A2 B2 C2 D2 E2 F2 G2 H2      B8 B7 B6 B5 B4 B3 B2 B1
A1 B1 C1 D1 E1 F1 G1 H1      A8 A7 A6 A5 A4 A3 A2 A1

        Figure 4.                    Figure 5
```

Recall also that A1 is bit 0 (MSB) of the occupied_squares bitmap, while H8 is bit 63 (LSB) of the occupied_squares bitmap. First, we rotate this left 90 degrees to make bits on a file adjacent to each other (note that the lower left-hand corner is always going to be numbered bit 0 in the 64 bit bitmap, and the upper right corner will always be numbered bit 63) as given in figure 5.

Next we rotate the original left 45 degrees to produce figure 6.

```
              H8                                 A8
            G8  H7                             A7  B8
          F8  G7  H6                         A6  B7  C8
        E8  F7  G6  H5                     A5  B6  C7  D8
      D8  E7  F6  G5  H4                 A4  B5  C6  D7  E8
    C8  D7  E6  F5  G4  H3             A3  B4  C5  D6  E7  F8
  B8  C7  D6  E5  F4  G3  H2         A2  B3  C4  D5  E6  F7  G8
A8  B7  C6  D5  E4  F3  G2  H1     A1  B2  C3  D4  E5  F6  G7  H8
  A7  B6  C5  D4  E3  F2  G1         B1  C2  D3  E4  F5  G6  H7
    A6  B5  C4  D3  E2  F1             C1  D2  E3  F4  G5  H6
      A5  B4  C3  D2  F1                 D1  E2  F3  G4  H5
        A4  B3  C2  D1                     E1  F2  G3  H4
          A3  B2  C1                         F1  G2  H3
            A2  B1                             G1  H2
              A1                                 H1

           Figure 6.                          Figure 7.
```

In this rotated bitmap, the bottom square (A1) becomes bit 0, and the rest of the bits above it fall into order behind that bit, ie A2 becomes bit 1, B1 becomes bit 2, continuing until H8 becomes bit 63.

Notice that in the above bitmap, the diagonal from H1 to A8, and all diagonals that are parallel to H1-A8 are now in adjacent bits. This is less convenient to use than the first two bitmaps, because we could easily compute how to shift the rank or file in question to the right-hand end of an integer to use as an array subscript, and also we knew to AND this shifted value with the constant 255 to remove all but the wanted eight bits. In the diagonals, we have varying lengths for the diagonals which means that the shift amount varies as does the mask to scrub off extra bits. Overcoming this problem is described later.

Next we rotate the original bitmap 45 degrees to the right to force the other diagonals into adjacent bits in the bitmap given in figure 7.

Once we have these, it becomes pretty obvious that we create three more arrays of bitmaps for attacks, in addition to the already discussed rank_attacks[64][64]. We now add file_attacks[64][64], diaga1h8_attacks[64][64] and diagh1a8_attacks[64][64]. And if we can discover some way to rotate the original occupied_squares bitmap, we can use the same array indexing scheme to turn the complex bitmap update procedure into two table lookups for bishops or rooks, and four for queens.

Later, after many discussions about schemes to rotate the occupied_squares bitmap, the simple solution was discovered. Rather than keeping just one occupied_squares bitmap and rotating it on demand (which we deemed nearly impossible to do efficiently), we would keep four occupied squares bitmaps, the normal one, plus three more rotated left 90 degrees, left 45 degrees and right 45 degrees. This is actually quite easy, because in procedure MakeMove() we find code like this for updating the occupied_squares bitmap:

occupied_squares^=set_mask[from]|set_mask[to];

That is the simple mechanism for making normal moves (captures, castling and en passant moves are handled slightly differently) because we know that the "from" bit in occupied_squares must be a 1, otherwise there would be no piece on this square to move, and we know that the "to" bit in occupied_squares must be a 0, unless this is a capture. So, by exclusive-ORing those two bits into occupied_squares, the from bit is cleared and the to bit is set.

The array of bitmaps set_mask[64] simply has a 1 bit set in the correct position for that element of the array. IE set_mask[0] has the left-most bit (bit 0) set, while set_mask[63] has the right-most bit (bit 63) set. To maintain the rotated bitmaps, we created three new sets of these mask bits, set_mask_rl90[64], set_mask_rr45[64], and set_mask_rl45[64]. Each of these takes a square number as the subscript, but turns it into the bitmap to set the appropriate bit in the rotated occupied_squares. For example, set_mask_rl90[0] would look actually have bit 7 set, because in the rotated_left bitmap, bit 0 becomes bit 7 when you look at the mapping.

This was the solution that we needed to make this viable. Rather than trying to rotate the occupied_squares bitmap, we now maintain all four of the occupied_squares bitmaps simultaneously, so that we can use two bitmaps(for bishops/rooks) or four bitmaps (for queens) as needed to produce attacks for that type of sliding piece.

For bishops, there is one clever optimization that can be used, although it is expensive in terms of memory. Recall that for rooks, we shifted the appropriate rank or file to the right-end of an integer, then we could AND this with 255 to extract the rank or file contents. And since every rank or file is eight bits long, this works well. But for diagonals, each one has a different length from its neighbor.

To make this algorithm simpler, we chose to continue to AND the diagonal with 255, which obviously gets the right diagonal, but which also gets some bits that are from an adjacent diagonal(s). If the diagonal of interest has only three bits in it, (a total of 8 unique states) we take each of these 8 states, and combine it with the 32 states of the extra bits. IE we now have the actual three bit state that is important replicated 32 times, so that no matter what the contents of the extra 5 bits, we will get the right attack bitmap for the three bit diagonal in question.

4. Computing attacks_from[64] using rotated bitmaps.

To generate attacks, we use 4 distinct arrays, one for the attacks along the ranks, one for attacks along the files, and one each for the two diagonals that pass through a square. We can call these arrays rank_attacks[64][64], file_attacks[64][64], diaga1h8_attacks[64][64] and diagh1a8_attacks[64][64].

To initialize these arrays (this is done when the engine is started, and then these arrays are treated as constants from that point on) we simply assume a sliding piece that slides in the direction being considered sits on the square (ie a queen). Then for each square, we initialize the rank_attacks[square][rank_contents] to the bitmap that matches the squares that a rook/queen on "square" could move to, assuming the rank is occupied as per the eight bit value "rank_contents". For example, take the occupied_squares bitmap used earlier, but add a rook on square F5 instead of a bishop as in the previous example (figure 8), and assume we are trying to initialize rank_attacks[37][100] (37 is the square F5, and 100 represents the occupied squares on that rank, 01100100). We initialize this entry in rank_attacks as shown in figure 9. This generates exactly the squares that a rook/queen would attack on the 5th rank, if the piece is on square F5 and the rank is occupied by the 3 pieces given above.

```
 - - X - - X X -     - - - - - - - -     - X - - - - X -
 X - - X X X X X     - - - - - - - -     X X X - - X - X
 - X X - - - X -     - - - - - - - -     X X - R X X X X
 - X X - - R - -     - - X X X - X X     - X - - - - X -
 X - X - - X - -     - - - - - - - -     - X - - - - - X
 - - X - - X X -     - - - - - - - -     X - X X X X - -
 - X - - X X - X     - - - - - - - -     - - X X - - X -
 X - - X - X X -     - - - - - - - -     - X - - X - - X

    Figure 8.           Figure 9.           Figure 10.
```

So far, so good. We can quickly access the rook attacks along a rank by doing this, but now we want to compute the attacks down the file, rather than across the rank. So we go to the occupied_rl90 bitmap which is an exact duplicate of the previous occupied_squares, only rotated to the left 90 degrees, as shown in figure 10.

When we extract the third 'rank', we now actually get the F-file's occupied status. We again use the precomputed attack bitmaps, but this time we use the attacks_file[37][251] (again, 37 for the square F5, while the occupied squares of this file are 11111011) entry which was initialized to the value shown in figure 11.

```
 - - - - - - - -
 - - - - - X - -
 - - - - - X - -
 - - - - - - - -
 - - - - - X - -
 - - - - - - - -
 - - - - - - - -
 - - - - - - - -

    Figure 11.
```

Now, by taking these two values, the attacks along the rank, and the attacks along the file, and ORing them together we get the complete set of attacked squares for a rook on F6 with this particular configuration of the file and rank contents. If this was a queen, we continue by using the two diagonal attack bitmaps and the two diagonal occupied_squares rotated bitmaps to get the diagonal attacks in the same way, and then we OR the diagonal attacks together with the file/rank attacks and we are done. Notice that there are no loops of any kind here. The algorithm for a sliding piece looks like this:

```
    BITBOARD attacks=0;
    if (BishopMover(piece)) {
      get diaga1 status (shift/AND);  /* 6 bits */
      attacks|=diaga1h8_attacks[square][status];
      get diagh1 status (shift/AND);  /* 6 bits */
      attacks|=diagh1a8_attacks[square][status];
    }
    if (RookMover(piece)) {
      get rank status (shift/AND);    /* 6 bits */
      attacks|=rank_attacks[square][status];
      get file status (shift/AND);    /* 6 bits */
      attacks|=file_attacks[square][status];
    }
```

And we are done, completely. The two tests, BishopMover() and RookMover(), return true if the piece slides diagonally (bishop or queen) or if the piece slides like a rook (rook or queen). In Crafty, this is encoded in the piece type, where P=1, N=2, K=3, B=5, R=6 and Q=7. Studying these numbers, if piece_type&4 is non-zero, it is a sliding piece. Then, if piece_type&1 is non-zero, this piece slides along diagonals and if piece_type&2 is non-zero, the piece slides along ranks/files.

5. Computing attacks_to[64]

The attacks_to[] bitmap was defined by Slate and Atkin as a bitmap with a one bit set for each square that attacks the target square. For example, attacks_to[28] (28 = E4) might look like figure 12, assuming E4 is attacked by a black rook at E8, a black knight at F6, and defended by a white rook at E1 and a white pawn at D3 (E4 is marked T for target).

```
 - - - - X - - -
 - - - - - - - -
 - - - - - X - -
 - - - - - - - -
 - - - - T - - -
 - - - X - - - -
 - - - - - - - -
 - - - - X - - -

    Figure 12.
```

In this figure "T" is the target square E4 and is really a 0 bit in the bitmap. The four X (1) bits indicate the four squares that have pieces attacking E4. If we want to know how many black pieces are attacking E4, we would simply AND this bitmap with the black occupied_squares bitmap and the result would have 1 bits on every square occupied by a black piece that attacks E4. Ditto for the white pieces attacking E4, where we AND this bitmap with the white occupied_squares bitmap as before.

The questions are (a) how can we compute such a bitmap and (b) is it very expensive to do so? Using the already described method of generating the attack bitmaps (recall that generating attack bitmaps for non-sliding pieces is trivial since all the squares a given type of piece attacks can be computed at program startup since this doesn't change like those of sliding pieces) we compute the bishop attacks for E4, then AND this with the bitmap of white/black bishops and queens, which gives us 1 bits for each bishop/queen that attacks this target square, regardless of the color. We save this result, and then repeat this for the rook moves and rooks/queens bitmap. Then we take the knight attack bitmap, AND this with the white/black knight bitmap, and repeat for the king and pawns. We end up with five bitmaps that when ORed together, enumerate every square that is attacking E4.

Now that we can produce a set of squares that a piece on "square" attacks (useful for generating moves) or a set of squares that attacks a specific "square" (useful for determining if the king is in check, for example) we now turn to using this information in a chess program.

6. Using bitmaps in a chess engine.

It is important to note that the above algorithm produces a complete attack bitmap for a single piece using no loops of any kind. However, in a chess program, these moves have to be turned into a set of from/to square moves so they can be tried, and this process is certainly going to end up being a loop, although the loop becomes quite simple. The "from" square is known, so we simply use the FirstOne() function to turn the first one bit into an integer "to" square and clear that bit. We loop until no bits are set. Here is some actual code to do just this for white queens: (note, in Crafty, a move is stored in 21 bits. The rightmost 6 bits are the FROM square, the next 6 are the TO square, the next 3 are the MOVING_PIECE, the next 3 are the CAPTURED_PIECE, and the final 3 are the PROMOTE_TO piece for pawn promotions.)

```
piecebd=WhiteQueens;
while (piecebd) {
  from=LastOne(piecebd);
  moves=AttacksQueen(from);
  temp=from+(queen<<12);
  while (moves) {
    to=LastOne(moves);
    move_list[i++]=temp+to<<6;
    Clear(to,moves);
  }
  Clear(from,piecebd);
}
```

At first glance, this seems like it might be pretty inefficient, but when studying a chess program and the move ordering issue, the first set of moves to be tried is the captures. To generate only capture moves, all we do is generate the above attack bitmap, then AND this with the bitmap containing all squares occupied by opponents pieces, and all that is left is the bits indicating which moves are captures. So to generate captures, which is the largest part of the chess tree, we only loop over the specific moves that capture opponent pieces, not having to loop over the empty squares between the sliding piece and the piece it can capture, and not having to do any specific testing to detect when we reach the edge of the board. This turns out to be a substantial savings when compared to traditional move generation. Here is the same code modified to produce only capture moves for white queens.

```
piecebd=WhiteQueens;
while (piecebd) {
  from=LastOne(piecebd);
  moves=AttacksQueen(from) & BlackPieces;
  temp=from+(queen<<12);
  while (moves) {
    to=LastOne(moves);
    move_list[i++]=temp+to<<6;
    Clear(to,moves);
  }
  Clear(from,piecebd);
}
```

It is clean and simple, and if there are no queen moves that capture an opponent's piece, the inner loop above is never executed at all. If you choose, you can pass your move generator a "target" bitmap. To generate all moves, pass it a target bitmap which is simply the complement of occupied_squares for the side on move, so that all moves (except capturing your own pieces) are generated. To only generate captures, simply pass it the occupied_squares bitmap for the opponent's pieces.

As has been previously shown, it is also simple to produce the attacks_to[] bitmaps when required. This is neither computationally expensive nor difficult to implement, and provides this functionality (attacks_to[sq]) if/when it is needed. The most common use for this type of information is to ask the question "is the king in check?" and that is even easier to do than previously described. For each type of piece (bishop/queen, rook/queen, knight, king and pawn) we generate the attack bitmap and then AND each with the opponent's piece bitmap only (we do not care if our pieces attack our own king, just whether the opponent's pieces do or not) and if any result is non-zero, we immediately return "true" saying "yes, the king is attacked and in check."

This can also be used to implement a static exchange evaluator to be used in move ordering. For a given target square, where a sequence of exchanges are to be evaluated, we first produce the attacks_to[sq] for that square, which enumerates every piece directly attacking the target. We find the least valuable opponent piece attacking this square by simply ANDing the attacks_to with each of six opponent piece bitmaps, starting with pawns, and going in order of piece value, king last. When we get a non-zero result from the AND, we know the piece value of the piece that would be used first to capture on the target square. We clear this attacker from attacks_to since he has been used, then we AND the remaining attacks_to with the six friendly piece bitmaps, to find the least valuable defender. We repeat until the attacks_to bitmap is empty.

But what about the sliding pieces that attack the target from behind a piece that is in front of it? This turns out to be easy to handle, also. We know the piece that we just "used" by clearing it from the attacks_to bitmap, so we can determine whether it is a sliding piece or not (ie a pawn, bishop, rook or queen can have pieces behind them in a "battery"). All we do, with another AND operation, is to see if there is another piece in the same direction behind the piece just used. If so, and it is the right type (bishop/queen for diagonals, rook/queen for ranks/files) then we simply OR that piece into the attacks_to, since it is now attacking the target square with the intervening piece out of the way. (We use the now-recognizable plusN[]/minusN[] bitmaps, and leave the simple 'how' as an exercise for the reader.)

Note that we are not considering anything but whether a piece attacks the target or not. It might be pinned on the king, pinned on a more valuable piece, or overloaded by defending two squares at one time. Errors are therefore possible. But notice that for the single target square, this is quite accurate, because having three pieces in battery (say a queen and two rooks) can be tricky if the queen is first, because we have to be sure we use the queen first in the swap analysis. Using this algorithm guarantees this since the rooks will not show up in the attacks_to until the queen is removed.

7. Conclusions

There are many other advantages for using bitmaps, but by far the most important is "data density". Notice that all 64 bits are important, even though most end up being zero, because they reflect the status of a particular square. This means that as a cpu moves these 64 bit bitmaps around, it is not wasting internal bandwidth by moving unused data, as opposed to what happens when a normal chess program is moved to a 64 bit machine. There, most of the 64 bit words are unused, and the unused parts don't represent useful information, they are just unneeded additional precision.

For the current 64 bit architectures (Alpha, HP, MIPS, to name a few) and for new 64 bit architectures such as the Intel Merced processor, bitmaps make a great deal of sense, because bitmaps take maximum advantage of internal processor register and bus size. For example, most of the normal bitmap operations (AND, OR, XOR, shift, and so forth) take at least two instructions on a 32 bit machine, one for each half of the 64 bit bitmap. On a 64 bit architecture, with no clock speed change at all, these operations run twice as fast, because they now require only one instruction to perform them. That is a performance gain that can not be overlooked, because it is completely free.

Many programs now use bitmaps for some functions, such as evaluating pawn structure (because it is a convenient and compact way to represent the pawns in a single variable), detecting whether a king is in check or not (because the bitmap makes it pretty easy to decide whether a sliding piece can attack the king using a single AND operation), and for other things. Chess 4.0 started the bitmap revolution by being the first program to use bitmaps exclusively for the board representation, and programs like Crafty (there are several others as well, too numerous to mention) have continued this development. Perhaps the primary performance improvement to using bitmaps came from the Crafty project however, when the concept of "rotated bitmaps" was developed as an alternative to the classic incremental update approach used in chess 4.0, because this produced a substantial performance increase with no loss at all in capability. The most interesting question now becomes "how else can bitmaps be used in a chess program to improve its performance even further?" There is great danger in misusing bitmaps, as the old proverb "to the man with a hammer, everything looks like a nail" warns. However, it is very important that we try (at least once) to hit everything we see to see if this "hammer" will work well, or show us that perhaps there is a better tool of another type for some tasks.

8. References

G. Adelson-Velsky, V. Arlazarov, A. Bitman, A. Zhivotovsky, A. Uskov, "Programming a computer to play chess," proceedings of the 1st Summer School on Mathematical Programming, Vol. 2, 216-252, (1969).

S. Cracraft, "Bitmap Move Generation in Chess," Journal of the International Computer Chess Association, Vol. 7, No. 3, pp. 146-153 (1984).

P. Frey, "An Introduction to Computer Chess," in Chess Skill in Man and Machine, P. Frey (ed.), Springer-Verlag, pp. 82-118 (1977).

E. Heinz, "How DarkThought plays chess," Journal of the International Computer Chess Association, Vol. 20, No. 3, pp. 166-176 (1997).

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