Yugoslav Journal of Operations Research
22 (2012), Number 1, 265-284
DOI:10.2298/YJOR081028011V

AN ALTERNATIVE EFFICIENT CHESSBOARD REPRESENTATION BASED ON 4-BIT PIECE CODING

Vladan VUČKOVIĆ
Faculty of Electronic Engineering,
University of Niš, Serbia
vladan.vuckovic@elfak.ni.ac.rs

Received: October 2008 / Accepted: May 2012

Abstract: This paper describes theoretical and practical aspects of an alternative efficient chessboard representation based on 4-bit piece coding technique. There are two main approaches used by the majority of computer chess programs: arrays and bitboards. However, after the years of researching and experimenting in chess engine *Axon* and its parallel version *Achilles*, we would like to introduce an alternative chessboard representation C. C. R. (*Compact Chessboard Representation*) based on a new coding technique that performs very well both on 32-bit and 64-bit hardware platforms.

Keywords: Computer chess, chess engines, data structures, chessboard representation.

MSC: 91 - Game Theory, 91-04, 91A24.

1. INTRODUCTION

The first major issue that must be addressed when we start to write a chess program is how to represent the chessboard and other supporting data structures that will be used in different parts of a chess engine. This primary decision, which is very often based on incomplete or insufficient information, has its consequences later, when we come to the main chess engine procedures like move generator or evaluator. At that point, one could discover that his data structures have some fundamental shortcomings and could not satisfy expectations about the efficiency or programming facility. Sometimes, these deficiencies can appear at the end of the development cycle when the engine is ported on some specific hardware platform. Such a revelation can be very unpleasant for the programmer because, at this stage, re-design of the basic data structure is not viable option anymore. Our intention, throughout this paper, is to present all relevant information about existing data structures in order to help even inexperienced chess engine programmers to properly select the basic data structure. Naturally, we will concentrate on rotated bitboards [4] as the most interesting modern chessboard representation. World Computer Chess Champion engine *Rybka* [5] is a best proof of efficiency of rotated bitboards concept. Also, as we mentioned, we will define a new data structure C. C. R. that was intensively tested and evaluated in another experimental grandmaster chess engine - *Axon*/*Achilles* [10],[12]. The C. C. R. could also be an alternative solution [12].

There are two basic board representations in use in contemporary chess programs. From the first research works about computer chess the most common data structure used by many chess engines is array board representation (in some papers referred as offset board representation). Shannon first mentioned this data structure in his fundamental paper back in the early 1950's [6]. After a long period of computer hardware development, especially in the area of 64-bit CPU-s, a new approach was discovered. The new chessboard representation data structure was called *bitboards* (a set of bit-vectors or bitmaps).

The first idea about the utilization of the bitboards could be attributed to D. Slate and L. Atkin, the authors of the famous Chess 4.5 engine [7], in the middle 1970's. They have described the approach of using twelve 64-bit unsigned integers, one for each type of a piece on the board. So, there are six bitboards for white pieces; pawn, knight, bishop, rook, queen, king and also six for corresponding black pieces. They annotated the connection between 64-bits in integer and the number of squares on the chessboard, so a 1 bit could be used to indicate the presence of a piece on a square and a 0 bit indicates the absence of a piece (empty square).

It should be noted that the *Kaissa* team [1] apparently developed this same idea independently of Slate and Atkin, approximately at the same time. After these first efforts, many other programmers experimented and used bitboards. However, the first noticed problem that remains till nowadays is that bitboards strongly require 64-bit registers and processors to run optimally. Also the problem of efficient generation of the supporting bitboards like attacking bitboards and others immediately occurred. Fortunately, that problem was successfully solved in Hyatt’s inspiring paper [4]. It is well known that prof. Hyatt is the author of the *Cray Blitz* (former World Computer Chess Champion) and *Crafty* chess engines where the utilization and efficiency of the rotated bitboard approach is demonstrated in a best way. From the other hand, the microcomputer revolution in the 1980's traced the new way of developing the chess engines. Generally, 64-bit mainframes were used only in a tiny proportion, so the dominance of the arrays as the basic chessboard data structure was prolonged. Array based chess engines were widely used in home, personal or specialized computers. Following the intensive development of the microprocessors and hardware, coupled with some important discoveries in software domain like null-move [3], chess engines achieved grandmaster strength.

This mainstream approach changed a few years ago after the appearance of the chess engine *Rybka*. In *Rybka*, potentials of the rotated bitboars are fully developed. High profile of the embedded expert chess knowledge in combination with 64-bit multicore implementation established *Rybka* as the best chess engine today. The brilliant performance in numerous computer vs. computer tournaments, including the World

Chess Championship and victories in all matches organized against top human grandmasters, prove this statement.

As we mentioned before, the main problem with bitboards could be defined as the imperative need for 64-bit CPU. For instance, the 32-bit test version of *Rybka* on AMD 64-bit CPU running on 2.4Ghz achieves 104 Knps (thousands of nodes per second). The same engine compiled with 64-bit compiler runs on the same CPU and operating system with 166 Knps that is 66% better result than 32-bit version. Such a difference can only be attributed to the difficulties in processing (additional executive code) of the 64-bit bitboards on 32-bit machines.

We hope that the data structure named Compact Chessboard Representation (C. C. R.) [9] that is in the main scope of this paper would satisfy this condition. Our experience with this type of chessboard definition is very positive. The experimental chess engine *Axon* contains C. C. R. as the main data structure requiring no 64-bit operating systems for the high performance.

In this paper we intend to offer a theoretical and practical solution to the aforementioned issues. There will be several sections dealing with various aspects of chessboard representation. After this initial assessment, our next section will describe some variations of the array chessboard representation. The third section will present details about bitboard chess representation including rotated bitboards. In the fourth section we plan to introduce our definition of Compact Chessboard Representation. The same section will discuss some problems of efficient generation of the attack data structures based on C. C. R. The fifth section will present some procedures connected with the C. C. R. implementation into the chess engines. At the end, we will try to briefly display the characteristics and utilization of all data structures we mentioned and our ideas for the future course of research.

2. ARRAY CHESSBOARD REPRESENTATION

The simplest way to represent a board is to create an 8x8 two-dimensional array. There are 13 different entities on the chessboard: 6 different pieces for white and black and an empty square. This implies that a byte (short integer) would be enough for one element representation. Each array element identifies which entity is occupying the square on chessboard. The next problem is encoding of these entities.

The most common approach is to consider zero as empty square, positive or negative values for white and black pieces respectively. Also, there are some other types of encoding but they have no specific effect regarding the efficiency. The first problem with an array-based approach arises from move generator procedure. For instance, if we want to generate all legal knight moves on the board we must check if the move is on the board or not. That could be done with two conditional instructions, or maximally 16 conditionals for all possible knights' moves. These could significantly slow down the move generation. However, the arrays are very obvious and simple data structures so that utilization of arrays as the basic structure results in reduced effort needed to implement different procedures. This is the main reason why this structure can be recommended to inexperienced chess engine programmers. But as far as efficiency is concerned one could require more sophisticated approaches.

2.1. 12x12 and 12x10 arrays

The first important improvement is to define one-dimensional array instead of two-dimensional. Someone could argue that machine definition of the two-dimensional array as well as any other data structure is one-dimensional. But, difference is in the data structure access. On two-dimensional array we must operate via two coordinates that generates at least one multiply command in executable machine code. Using the one-dimensional array eradicates need for any multiplies. A chess program frequently accesses the arrays so the savings could be huge. Nevertheless, the problem about the determination of the legal moves in move generation still remains, although it is simplified.

Next development proposes to have the chessboard represented not at an 8 x 8 array but at a 12 x 12 array. The 8 x 8 array of the chessboard is centered with a 2-rank border around it. For these purposes, the 12x10 array could also be used, but it is not reflected on our observations. This expanded array ensures that all moves generated by sliding or non-sliding pieces lie within the array. All knight moves also lie within the array, no matter where it stands. The program initiates the 2-rank border as "filled" using some pre-defined constant and thus, the moves into the border by any piece would be illegal. In combination with one-dimensional representation this method is very sufficient for the move generator because boundary checking is reduced to test if the destination array element is "filled" or not. This eliminates array coordinate calculation of any kind so the access is maximally accelerated. We should disclose that the very first version of *Axon* chess engine, using 12x12 one-dimensional array for chessboard representation obtained the international master chess playing strength (ELO above 2550).

2.2. Other Chessboard Representations

There are some other chessboard representations but all of them are useless for the chess engines although they find their implementation in some other applications. For instance, Forsyth-Edwards (FEN) chessboard definition is used intensively by chess programs for saving chessboard positions to external storage in ASCII format in a single line of text. Also a human may view and decrypt that information easily.

Our next example is Huffman encoding scheme that allows a complete board state to be represented in just 23 bytes. The main idea of this interesting approach is corresponding with ideas for general data compressor engines (ZIP, RAR, ARJ). The most frequent chessboard elements are coded with a fewer bits than less common ones. For instance, the empty square is coded with one bit 0, the pawn is coded with two bits 10b etc. Huffman encodings are rather processor intensive. The other board representations, including classic array representation that has been mentioned before, try to minimize required processor and memory resources. The compressed chessboard is very well suited to storage of long-term chess knowledge especially in storing positions in an opening book. Using the Huffman technique the millions of an opening chess positions in tablebases could be compressed with a very high ratio. Also, it could be also used in transposition tables for shallow entries. There are some other coding schemes and chessboard representations but they are marginal and could not be interesting for usage in an efficient chess engines.

3. BITBOARD REPRESENTATION

Bitboards are the chessboard representation based on a circumstance that a chessboard has 64 squares that is exactly the capacity of one long integer. The trend in modern CPU-s, as well as in old mainframes, is to use exactly 64-bit integers and data structures. In that way, bitboard have characteristics that make them especially attractive for computer chess applications [2]. One 64-bit register is able to represent the Boolean condition for each square of the chessboard. Those conditions could define piece replacement on the chessboard as well as some other information useful for chess engine operation, like attacking matrices (attacking matrix defines which squares are attacked from a piece on a specific square). Among many advantages of bitboards we could emphasize that Boolean operations can be performed on all squares in parallel [13],[17]. Nevertheless, the disadvantage is that programming, maintaining and utilization of the bitboards in chess engine is more complex compared to array approach. Also, the bitboards run significantly slower on 32-bit machines [14],[15]. Furthermore, updating all bitboard information after each move can be costly and that is especially visible in a case of attack table generation.

If we accept that each bit in a bitboard indicates the absence or presence of some state about each place on the board, a board position can then be represented using a series of bitboards. Following the fundamental work of Slate and Atkin there must be minimally 12 bitboards for each side and piece type. In practical computer chess programming the requirement for the other types of information that must be stored and computed efficiently is imminent. We have named attacking matrices (bitboards) for each piece, but very often we need to have bitboards for some other piece status. In that way, the total number of bitboards that have to be maintained in chess engine amounts to approximately twenty.

3.1. Attack Bitboards

The attack bitboards are widely recognized as being advantageous for the move generator, evaluator or any other procedure where the influence among pieces is concerned. The *attacks to* bitmap was primary defined by Slate and Atkin [7] as a bitmap with a one bit set for each square that attacks the target square. Using this definition, it is obvious that attack bitboards must be recalculated from the scratch in each node of a tree search. Practically, we must use some of the 12 piece replacement bitboards to generate attack bitboards. The very first efforts in that direction have shown that this approach is slow and unpractical for real chess applications. Fortunately, there is another approach, named rotated bitboards, which solves the problem [4]. Using that approach, bitboards became very useful and efficient way to represent chessboards and other supporting information in chess engine procedures.

3.2. Rotated Bitboards

The elegant solution is found in a simple variation of the normal bitboard using a 90-degree rotated occupied bitboard. In this bitboard each file of the chessboard is represented by one byte. This 90-degree rotated bitboards are maintained in the same way as the basic bitboards. The rook attacks across a file can be determined now through the rotated bitboards. In the same way, 45-degree left and right rotated occupied bitboards are introduced, where a left or right diagonal will be stored in one byte. The main diagonals A1-H8 and H1-A8 are stored in a full byte and other diagonals content less than one byte. This 45-degree rotated bitboards are also maintained in the same way as the basic bitboards. Using these bitboards and a lookup table we can determine bishop attacks. Queen attacks are generated by combination (OR operation) of the rook and bishop attack bitboards. Finally, the whole system is completed.

By adding two rotated bitboards, we are able to efficiently generate attacks for any kind of sliding or non-sliding pieces from lookup table without using slow loops. These theoretical improvements in combination with 64-bit CPU architecture established bitboards as the dominant chessboard representation today.

4. COMPACT CHESSBOARD REPRESENTATION

As has been previously emphasized, with a full respect to their elegancy, the bitboards have some flows that limit their performances in some conditions.

First of all, bitboards are substantially slower on 32-bit machines than on 64-bit. This limitation is impossible to overcome because the bitboards require compact 64-bit CPU registers to operate with maximal efficiency [16]. The division on two 32-bit integers produces variety of problems to compiler and resulting executable code cannot be efficient enough.

The second limiting factor is that programming the bitboards is much more complicated than programming the arrays. This fact could be confirmed by any programmer who worked with both structures. The possibility for generating bugs in code is higher, too. The maintenance of a bitboards source code is also more complicated. Also, a bitboard system (including the occupied and rotated bitboards) has between twelve and twenty 64-bit integers to manage at each node of the tree search. In quiescence search, where efficiency is on primary focus, this huge number of bytes which must be transferred from one node to another deep in a search tree could be uncomfortable ballast. These flaws are much more exposed when we use machine code instead of C for chess engine programming.

This situation is very well documented in the *Axon* development [16]. The experimental chess engine *Axon* is written in x86 assembly language and manually coded (about 30000 lines in assembly). The first versions of the *Axon* have used 12x12 array chessboard representation. Later, the engine development was directed towards 32-bit environment under *Windows XP* operating system. In these circumstances, the need for a new chessboard representation that will be suitable for the low level programming was very pronounced. The new data structure had to be more compact and efficient than arrays and accommodated for the 32-bit CPU-s. On that point, the 64-bit adapted bitboards where excluded as the option. So, something new had to be invented.

4.1. 4-bit Piece Coding

In order to explain the data structure better, let us propose a simple type of piece coding. Also, let us emphasize that the form of piece coding is completely irrelevant for our further analysis. However, adequate piece coding could be beneficial for the efficient move generator realization. The chess board hosts 12 different pieces, six white and six black pieces. Including the empty square there are 13 entities. We need four bits to represent all of them (2^4=16 combinations). The piece coding can be done in the following way:

Table 1: A sample of piece coding

PIECE	**CODE**	**Dec.**	**Hex.**
Empty square	0000	0	0
White pawn	0001	1	1
White knight	0010	2	2
White bishop	0011	3	3
White rook	0100	4	4
White queen	0101	5	5
White king	0110	6	6
Black pawn	1001	9	9
Black knight	1010	10	0A
Black bishop	1011	11	0B
Black rook	1100	12	0C
Black queen	1101	13	0D
Black king	1110	14	0E

This piece-coding scheme is used in *Axon* chess engine. The black pieces have most significant bit set and the same structure of low significant bits as white pieces. There are many variations of this table implemented in different chess engines. Nevertheless, we could conclude that 4 bits (one half-byte) is the minimal uncompressed form of one piece/empty square coding. There are 64 squares on chessboard, so we need 32 bytes to define it completely.

4.2. C. C. R. Definition

As we know, there was no serious effort to define compact chessboard representation idea theoretically and prove it in practice. One of the reasons that could be mentioned is that high performance compact chessboards can be realized strictly in machine code (assembly). Only in this case a new data structure is able to be fully beneficial to overall chess engine performance. According to our previous definition of the minimal piece coding with 4 bits per square, an entire rank can be represented by one 32-bit register. Of course, there will be additional registers for remaining position information. Having in mind these facts, we could define a data structure containing only eight 32-bit machines registers representing the whole chessboard. This definition could be represented by the following line in *Pascal:*

```
CCR:  array [0..7] of cardinal;   {instead of cardinal someone
                                        could use 32-bit integer}
```

This data structure is enough to define chessboard and it also illustrates the simplicity of a compact structure. Thus, the whole chessboard is compressed to 8x4=32 bytes of memory. For instance, minimal configuration of bitboards (without rotated bitboards) contains 12x8=96 bytes of memory. The manipulation over that compact data structure is very effective. We have only 32 bytes (four 64-bit registers) to manage in a chess tree search. As we will show, the compact representation has some attributes of bitboards as well as of arrays. Also, using a jump table, we can go directly to the machine procedure to generate moves for any type of piece or evaluate its value. Using the register rotation method no checks for the edge of the board are required increasing move generation speed. There are some other important features that make the compact chessboard representation very interesting choice for the high performance chess engine realization.

4.3. Move Generation Based on C. C. R.

The data structure that we propose is applicable to any kind of search algorithms and procedures. The first one we intend to mention is move generator. Move generator is a procedure that creates a list of legal or pseudo-legal moves [16],[18]. Legal moves are generated strictly according to the rules of the game of chess. Pseudo-legal moves could be illegal mostly in open-check situations. The legalization of the pseudo-legal moves is postponed to the search procedure. Now, we will consider the pseudo-legal C.C.R move generator. Next figure shows the test position on a chessboard:

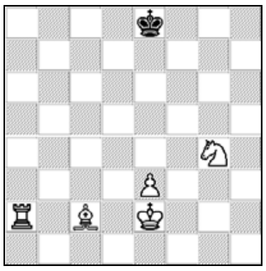

Figure 1: Test position. White is on the move.

According to piece coding scheme shown in Table 1 we initiate this position matrix:

Table 2: Position matrix generated from the chess position in Figure 1

0000	0000	0000	0000	**1110**	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	**0010**	0000
0000	0000	0000	0000	**0001**	0000	0000	0000
0100	0000	**0011**	0000	**0110**	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000

Table cells are in binary half-byte format *(nibble)*. Each rank represents one element in C. C. R. data structure so that we have following C. C. R.:

```
CCR(+0):    0000E000h
CCR(+4):    00000000h
CCR(+8):    00000000h
CCR(+12):   00000000h
CCR(+16):   00000020h
CCR(+20):   00001000h
CCR(+24):   40306000h
CCR(+28):   00000000h
```

The values are in hexadecimal format. As we already mentioned, C. C. R. contains eight 32-bit integers. We have specified offset in bytes for the each C. C. R. element from the beginning of the data structure in the brackets.

Finally, our compact representation of the position in Figure 1. could be realized as a list of eight decimal integers:

```
CCR = (57344,0,0,0,32,4096,1076912128,0).
```

Also, we could combine two 32-integers in one 64-bit and represent the position through only four 64-bit registers. These formats are perfectly suitable for both 32-bit and 64-bit CPU-s. Also, the existing 32-bit and 64-bit processor registers, including MMX registers, could hold several C. C. R.-s at the same time, allowing huge possibilities for additional optimization and manipulation inside the CPU core without use of the operational memory. By using the diagram in Figure 1. and the following numerical values, we will first consider the move generation of the one non-sliding piece – white knight. The possible moves of the white knight are presented in the following diagram:

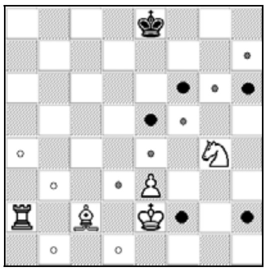

Figure 2: Legal moves of the white knight and bishop

Steps in the procedure of generating the pseudo-legal moves (that does not include the maintaining of the open-check) for knight using the C. C. R. could be abstracted in the following way:

First, we define displacement and the value of the corresponding C.C.R rank. For white knight, displacement is +16 and a value is 00000020h. Further, we create rank mask. It contains nibble 1111 binary (F hexadecimal) at the piece position. The rank mask is 000000F0h.

The move generation could start now. There is a maximum of eight legal moves for the knight. If the knight has the coordinates (h,v) the legal moves are at coordinates (h-2,v-1), (h+2,v-1), (h-1,v-2), (h+1,v-2), (h-2,v+1), (h+2,v+1), (h-1,v+2), (h+1,v+2).

Having in mind C. C. R. organization, it is obvious that vertical increments/decrements are realized as displacement calculation. The previous (upper) rank has current displacement decreased by 4. The next rank has displacement increased by 4. In our case, the upper rank has displacement +12 and the following rank +20. Horizontal movements are generated using the register shift to left (for decreasing) or to right (for increasing) by 4 bits. Finally, the move could be added to the move list if the logical AND operation between C. C. R. mask and dynamic rank mask generated by displacement changing and rotation, is zero. The zero result ensures that there is no bit set in a corresponding position in C. C. R. concluding that the square is empty.

In Figure 2. the white knight is posted at G4. To check if the squares E3 or H6 are empty, it is enough to execute a few machine operations:

```
For E3:  if (CCR(+20) and (MASK shl 8)) = 0 then move_G4_E3 is legal;

For H6:  if (CCR(+8) and (MASK shr 4)) = 0 then move_G4_H6 is legal;
```

The label MASK hashes the value 000000F0h in our example. There are maximally eight of these conditions for every possible knight direction. The primary question here is the bound control. Fortunately, for the C. C. R. this task is not tough. Vertical bound control is managed by displacement checking. If a displacement goes under +0 or above +28 the further code could be skipped. In machine language, checking for negative displacements is automatic, using the S (sign) flag bit. For the upper bound, one extra compare command (CMP) is needed. The situation for the left/right bounds is even simpler.

The rotation to the left or right automatically resets dynamic mask to zero if it goes out of bounds. Zero flag (Z) simultaneously gets the value 1 performing the conditional machine jump. It is easy to notice that all input data are 32-bit integers, enabling the efficient optimizations for 32-bit processors if high programming languages (C or Pascal) are employed.

The second non-sliding piece is king. There are also maximally eight legal moves for the king. If the king is posted at coordinates (h,v) the legal moves are at coordinates (h-1,v-1), (h,v-1), (h+1,v-1), (h-1,v), (h+1,v), (h-1,v+1), (h,v+1), (h+1,v+1). The king moves generation procedure is analogue to the knight's one.

The next table (Table 3) shows the displacement and dynamic mask rotations layout for the king depending on a direction.

Table 3: Displacement and rotation layout

displacement:=displacement-4 mask:=mask **shl** 4;	displacement:=displacement-4	displacement:=displacement-4 mask:=mask **shr** 4;
mask:=mask **shl** 4;	**THE CURRENT PIECE POSITION**	Mask:=mask **shr** 4;
displacement:=displacement+4 mask:=mask **shl** 4;	displacement:=displacement+4	displacement:=displacement+4 mask:=mask **shr** 4;

For pawns, the advance moves are treated simple. If the white pawn is posted at coordinates (h,v) the advance legal moves are at coordinates (h,v-1) and eventually (h,v-2) if the pawn is at the origin position on the second rank. The capture moves are also simple to check: (h-1,v-1) and (h+1,v-1). The extra code is needed for the special pawn actions like promotion or en-passant.

The operations for the sliding pieces are similar in fact. For instance, we will consider generating moves for the white bishop posted on C2. Let us presume that we want to generate moves on the diagonal C2-H7. Analogue to our previous consideration about the knight, we will generate the dynamic MASK first. For bishop mask will be 00F00000h and displacement is +24. Diagonal moves could be generated using the simple loop:

```
repeat
   MASK = MASK shr 4;
   displacement = displacement -4;
   edge := (MASK=0) or (displacement<0) or
   ((CCR(displacement) and MASK)<>0);
if not edge then GENERATE_MOVE;
until edge;
```

The Boolean variable *edge* gets value true when the horizontal/vertical bounds are reached or a piece in a line of sight was discovered. All other diagonals could be treated at the same why by implementing rotation or displacement calculation according to table 3. Thus, there are four loops in sequence for all 4 diagonals. Situation for the rooks is even simpler. For instance, the file moves upper from the current rook position could be generated using the loop:

```
repeat
  displacement = displacement -4;
  edge :=  (displacement<0) or ((CCR(displacement)
  and MASK)<>0);
if not edge then GENERATE_MOVE;
until edge;
```

Dynamic variable MASK has constant value. The rank moves generation to the right is also simple:

```
repeat
  MASK = MASK shr 4;
  edge :=  (MASK=0) or ((CCR(displacement) and
  MASK)<>0);
if not edge then GENERATE_MOVE;
until edge;
```

In this case displacement is constant. For the rook moves, there are two loops for files and two for ranks. To generate the moves for queens we merge procedures for bishops and rooks.

4.4. Generating Attacks

The first proceedings in computer chess indicated that the pure position definition is not enough to build efficient chess engine procedures like evaluator or plausible move generator.

On the other hand, the extra data structure needs extended code for generating and maintaining. One of the fundamental principles in computer chess is that every heuristic drops down the NPS (node per second) factor. The question that still remains unanswered is how to find the best balance between the speed and knowledge of the chess program.

One widely recognized data structure for supporting the heuristics in chess engine is *attack tables* or *attack bitmaps*. They are defined as the data structures, mostly with same format as chessboard representation, implementing the influence of the pieces to other pieces or empty squares. The nature and format of those tables are different, but there are many common features in their implementation in different chess engines.

For instance, the *attack_to* bitmap was defined by Slate and Atkin as a bitmap with one bit set for each square that attacks the target square. This information is useful both for evaluator and heuristic move generator.

At each node in tree search, we have to compute a set of attack bitmaps for all 64 squares. For each square, there are 256 different rank states (we have mentioned before that each rank is defined by 8 bits in bitboard). Of course, this information could be pre-computed and stored in appropriate hash table to accelerate the attack generation. With some improvements, the size of this hash table could be reduced to array[64][64]. This analysis is related to rank attacks but could be applied to all other such arrays.

If we use the technique of rotated bitboards we need to use four distinct arrays, one for the attacks along the ranks, the other for attacks along the files, and one each for the two diagonals that pass through a particular square. To initialize these arrays we presume that a sliding piece which slides in the certain direction occupies the particular square. This is done once when the engine is started and then these arrays are treated as a constant hash values.

Using the C. C. R., generation of the attack tables is also simple. The following method is elaborated, implemented and intensively tested in *Axon* chess engine. In *Axon*, the attacks are coded: for pawn 1000b, for knight and bishop 0100b, for rook 0010b and for queen and king 0001b. This descending order is very important to determine the influence on a particular square or attacks on some piece.

Now, we pass once through the C. C. R. and generate attacks from all pieces in all directions according to their legal movements. In fact, the procedures are exactly the same as in pseudo-legal move generator. A pawn generates attack diagonally on two nearest (capture) squares. This elegant solution is possible because all structures are compact. The addressing of the attack structures is done only by the changing the displacements. For instance, we will present the state of all three arrays for the position in the following figure:

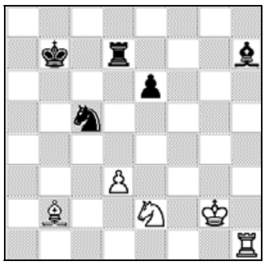

Figure 3: Illustrative position for attack generation

Using the previously realized coding, the contents of the matrices could be represented in a following sequence (elements are in binary numeric system, first column represents offsets):

Table 4: C. C. R. definition of the position

Off.								
+0	0000	0000	0000	0000	0000	0000	0000	0000
+4	0000	**1110**	0000	**1100**	0000	0000	0000	**1011**
+8	0000	0000	0000	0000	**1001**	0000	0000	0000
+12	0000	0000	**1010**	0000	0000	0000	0000	0000
+16	0000	0000	0000	0000	0000	0000	0000	0000
+20	0000	0000	0000	**0001**	0000	0000	0000	0000
+24	0000	**0011**	0000	0000	**0010**	0000	**0110**	0000
+28	0000	0000	0000	0000	0000	0000	0000	**0100**

Table 5: WHITE_ATTACKS definition

+32	0000	0000	0000	0000	0000	0000	0000	0100
+36	0000	**0000**	0000	**0000**	0000	0000	0100	**0010**
+40	0000	0000	0000	0000	**0000**	0100	0000	0010
+44	0000	0000	**0000**	0000	0100	0000	0000	0010
+48	0000	0000	1000	0100	1000	0100	0000	0010
+52	0100	0000	0100	**0000**	0000	0001	0101	0011
+56	0000	**0000**	0000	0000	**0000**	0001	**0000**	0011
+60	0110	0010	0110	0010	0010	0011	0111	**0001**

Table 6: BLACK_ATTACKS definition

+64	0001	0001	0001	0010	0000	0000	0100	0000
+68	0001	**0110**	0011	**0100**	0010	0010	0010	**0110**
+72	0101	0001	0001	0010	**0100**	0000	0100	0000
+76	0000	0000	**0000**	1010	0000	1100	0000	0000
+80	0100	0000	0000	0010	0100	0000	0000	0000
+84	0000	0100	0000	**0110**	0000	0000	0000	0000
+88	0000	**0000**	0000	0000	**0000**	0000	**0000**	0000
+92	0000	0000	0000	0000	0000	0000	0000	**0000**

Sliding pieces (rook and bishop in our case) stop at the first occupied element in C. C. R. in the line of sight. The method of attack generation is very suitable for different kind of optimizations using the circumstance that a whole C. C. R. rank may be loaded into the single 32-bit register. Horizontal attacks (for rooks and queens) could be generated only by one OR machine command. Vertical attacks need a small loop. Bishop diagonal attack generation could be optimized using the double mask (two diagonal squares could be simultaneously set). For queens, a triple mask could be used; for instance, one vertical and two up-diagonal attacks could be set simultaneously. In practice, these three structures are sufficient to support quality move generator and evaluator but a greater number may be advantageously employed.

The task: "indicate attacks on white pawn on D3" is solved through the following procedure: "Look at BLACK_ATTACK. Third rank in C. C. R. has offset +20. Displacement for corresponding BLACK_ATTACK rank is +20+32+32=+84. Mask is same as at C. C. R. : 000F0000h. So, BLACK_ATTACK(+84) AND 000F0000h. gives 00060000h. Number 6 is 0110b in binary representation so the white pawn is attacked minimally by two black pieces, one light piece and rook.". The conditions like these, which are very often in concrete programming, need only one or two 32-bit machine command and one memory access to comply without need for any default hash tables. According to our experience, compared to bitboards, the C. C. R. is much easer for a programmer to handle. Of course, this is the basic definition so some shortcomings exist. For instance, if the square is attacked by a light piece, there is no way to find out if it is attacked by bishop or by knight. Also, multiple attacks from the same kind of pieces (like rook doubling) are not verified. The general solution for these and any other similar questions that could arise in the future of engine development is also elegant. The common situation is that chess engine programmer needs a new data structure according to some new requirements that could not have been predicted in the early stage of development.

The new structures do not obstruct previous definition. The access remains efficient, just displacement is different. The dynamic masks used to operate on C. C. R. can be used unchanged for masking the other data (like attacks). The every new symmetric data structures add 4-bits for white and 4-bits for black at every square. The programmer can use these new 8 bits as he specifies. Also, there is no limitation in the number of future structures that could be added to extend basic C. C. R. definition.

5. USING C. C. R. IN CHESS ENGINE PROCEDURES

In this section we will briefly address some important issues concerning implementation aspects of C. C. R. Theoretical inventions we described had been evaluated after a long period of testing. Our practical observations support the theoretical views analyzed before.

5.1. Move Generator

The move generator procedure generates a list of pseudo legal moves using the C. C. R. The main idea is to use two concentric loops to pass thought the data structure. Outside loop controls displacement ranks and inside one controls the rank rotation serving the 4-bit index for table jump:

```
for d:=0 to 7 do
begin
   rank:=CCR[d];
      for r:=1 to 8 do
      begin
           CALL(jump_table[rank and 1111b]);
           Rank:=rank shr 4;
      end;
end;
```

Each piece has its corresponding procedure handler. The procedure addresses are loaded into the jump table. Using the 4-bit piece code encapsulated in C. C. R., this procedure uses efficient table jump that is easy to realize in machine code. In that way the usage of the slow case structure that is typical in this kind of procedures is avoided. All piece procedures use the same data structure (only eight 32-bit registers for C. C. R. description) improving the internal CPU optimization.

5.2. Evaluator

The evaluator uses the same procedure as the move generator but with different jump table [19],[20]. As the presumption for the evaluator, the attack matrices have to be generated first. Also, their procedures use the same format as previous procedure. In *Axon* engine core, all mentioned procedures are integrated into one. Using the same code frame, with C. C. R. as the common input data structure, this procedure changes the lookup table depending of input values and performs any mentioned function. Evaluator also uses the WHITE_ATTACKS and BLACK_ATTACKS. One detail, that could not be neglected, is that one must reset the attack arrays before evaluating the position or sorting moves by heuristic weights. This could be done using the sixteen 32-bit machine clear commands or more efficiently by eight 64-bit **movsq** MMX commands. The scanning for a piece in a rank is almost as efficient as for bitboards by using the machine **BSR** or **BSF** command. After the bits scan operation it is enough to perform one AND 1111b operation to extract the file number.

5.3. Quiescence Search Procedure

The critical point in chess engine is its quiescence search procedure. Depending on quality of knowledge embedded in quiescence procedure, it consumes almost 50%-70% of total time in tree search [21]. The quiescence procedure implements evaluator, heuristic search (based on captures, special checks, promotions ets.) through the classic Alpha-Beta algorithm.

The quiescence search is a recursive procedure and constantly calls itself with a new position on the stack. In our case, the position definition using C. C. R. has only 32-bytes. The other mentioned structures like rotated bitboards or classic arrays have substantially more bytes to store to deeper levels. Also, C. C. R. is used as input by evaluator and move generator. In that way, overall savings in byte transfers through the quiescence procedure are significant comparing to the other data structures. Combined with machine-coded procedures, all conditions for extra fast quiescence procedure are accomplished.

5.4. Generation of the 64-bit Transposition Keys Without Zobrist Method

Most of the recently available chess engines have procedures developed on the basis of the classic approaches using transposition tables. The addresses and keys for those tables are 64-bit by default. The widely recognized method for generating such keys is a *Zobrist* procedure [11]. Using the 64-bit Zobrist key instead of full positional matrix is also the probably most popular method for draw detection. This system implies the usage of random numbers in a phase of program initialization as well as storing and processing the keys changes in move forward procedure. This results in a rare collision problem. If we want to realize quiescence search procedure without a hashing at all, we can implement variant string method [8]. However, using the C. C. R. we have another effective possibility. In Axon [9], this procedure works independent of any external hash function, uses available data structures in quiescence and runs fast. Employment of primary numbers as a rotation base is a rule of a thumb but, surprisingly, performs very well without collisions. This procedure has been intensively tested on a hundreds of billions positions in real time.

5.5. The Efficient C.C.R Conversion to Bitboard

We have already stressed that bitboards reach very high performance on 64-bit CPU and demonstrated many useful features of the C. C. R. in the previous elaboration. The provoking question is if it is possible to merge these two data structures into the same chess engine?

Fortunately, there is a possibility to design very efficient conversion procedure from the C.C.T. to classic or rotated bitboards, optionally. This procedure implies the usage of the 64-bit environment. It follows the same idea as C. C. R. move generator and evaluator. The piece of code in C. C. R. is used to address corresponding element in a bitboard structure. Let us first define data structures. There is the complete bitboard set defined in the following code:

```
type
tbitboards = record
      bempty: int64;
      bwhitepawn: int64;
      bwhiteknight: int64;
      bwhitebishop: int64;
      bwhiterook: int64;
      bwhitequeen: int64;
      bwhiteking: int64;
      bempty: int64;
      bempty: int64;
      bblackpawn: int64;
      bblackknight: int64;
      bblackbishop: int64;
      bblackrook: int64;
      bblackqueen: int64;
      bblackking: int64;
      bempty: int64;
end;
```

```
var
       CCR:   array [0..7] of cardinal;
       BITBOARDS:    tbitboards;
```

The bitboard definition could also be represented as the array of 64-bit integers:

```
       BITBOARDS: array [0..15] of int64;
```

The order of bitboards must strictly follow the contents of Table 1. The conversion procedure does not contain *if* or *case* structures:

```
procedure conversion;
var x,y,rank,adr:cardinal;
begin
    for y:=0 to 7 do
    begin
        rank:=CCR[y];
        for x:=0 to 7 do
        begin
          adr:=rank and 1111b;
          rank:=rank shr 4;
          biboards[adr]:=bitboards[adr] or (1 shl ((y shl
3)+x));
        end;
    end;
end;
```

This code is efficient enough running in High Level Language (C++). But, the machine code equivalent to this procedure is extremely fast and it could be used whenever we want to commutate C. C. R. to bitboards. For instance, we could use C. C. R. for tree and quiescence search including move generation and flip to bitboards to evaluate position. This hybrid approach eliminates the necessity of the ponderous rotated bitboards structure handling through the tree search, keeping the efficiency of a bitboard based evaluator. We prefer this complementary approach as the most valuable one for the further research.

6. EXPERIMENTS AND EVALUATION

For the evaluation of the C.C.R. technique that is inseparable from the support algorithms that are shown in paper, we use two methods: automate testing games against other programs that use standard coding techniques chessboard and direct matches against professional chess players.

6.1. Automatically matches against other computers

Using standard graphical environment that supports UCI protocol it is possible to arrange automatic matches between experimental versions of some standard professional and amateur program. The entire organization is fully automatic matches in the GUI. In this way, we provide the exact same conditions for both participants in the match so that the results, especially if a large number of parties have played, are a very authoritative. Given all the advantages of the automatic matches for the analysis of complete programs, this type of test cycles in modern chess development program became very popular. An example of automatic generation of reports is shown in the following datasets:

```
  Program                  Elo    +   -   Games   Score   Av.Op.  Draws

1 Achilles               : 2560  111 169    22    86.4 %   2240   27.3 %
2 Fruit_21               : 2240  169 111    22    13.6 %   2560   27.3 %

   Program                 Elo    +   -   Games   Score   Av.Op.  Draws

1 Achilles               : 2459   93 104    40    66.2 %   2341   27.5 %
2 Shredder 9 UCI          : 2341  104  93    40    33.8 %   2459   27.5 %

   Program                 Elo    +   -   Games   Score   Av.Op.  Draws

1 Achilles               : 2463  120 131    26    67.3 %   2337   26.9 %
2 Aristarch 4.50         : 2337  131 120    26    32.7 %   2463   26.9 %
```

These test suits show the advantages of the Achilles (parallel version of C.C.R. Axon) against 3 strong chess engines with a high level of victory percentages.

6.2. Matches against chess masters

Matches and tournaments in which the program is under the same conditions of strong play against chess masters are the most important method of testing. Unfortunately, the events of this type are rare, so this method can not be regarded as the standard method of testing.

So far, the program Axon played many tournaments and matches with very good results, whose details are presented [9]. The parties have played most of the rapid pace of 15 minutes per player. The program has achieved many victories against chess masters with FIDE ratings. In particular, pick out a few victories against international masters, among which is the strongest opponent, whom he defeated in the tournament conditions, had a rating of 2440 ELO points [9].

Of course, the biggest success of the program, can be distinguished by a two-day match against grandmaster Igor Miladinovic, held at the Faculty of Electronics in Nis, in July 2007. Computer won with a score of 3.5 to 0.5. Details of the match are on the site: http://axon.elfak.ni.ac.rs. [22].

7. CONCLUSION

We have proposed a novel definition of a chessboard representation using Compact Chessboard Representation. Definition of the new data structure and explanation of the key chess engine procedures based on C. C. R. are also presented.

Our primary aim has been finding a solution for the problem of low efficiency of the rotated bitboard system on 32-bit machines, especially in quiescence search routines. Also, the new data structure, that has some useful characteristics compared with bitboards as well as arrays, provides optimal performance on a variety of hardware and software platforms. We should emphasize that a performance peak could be expected in complete machine coded chess engine cores like *Axon* chess engine. The efficiency of the new approach is most visible in the realization of a fast quiescence search routine.

As we already stated, the efficiency of the different data structures is difficult to assess since there are many access points to the structure in chess engine. The only way to determine a real value of the new approach is to implement it and to check the real-time performance. We hope that the C. C. R. based *Axon 4.0* experimental chess engine also with its grandmaster strength parallel version (*Achilles*) demonstrates the worth of the new ideas [22]. Our long-term experience both with rotated bitboards and C. C. R. also affirms these statements.

We also plan to inquire a complementary implementation of the bitboard and C. C. R. in the same chess engine due to the fact that there is a very efficient cross-conversion procedure (Section 5.5). Also, a very interesting idea is to apply rotated method to the C. C. R. itself, analogue to bitboards. This could open a new and very promising field of research. We are sure that there are many possibilities for the further speedups. We could conclude that Compact Chess Representation has characteristics that make it especially attractive for the high performance chess engines. In that way, we provided a further impetus to the computer chess data structures development.

Acknowledgements: The work presented here was supported by the Serbian Ministry of Education and Science (project III44006).

REFERENCES

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

[2] Cracraft, S.M., “Bitmap move generation in chess”, *ICCA Journal*, 7 (3) (1984)146-153.

[3] Donninger, C., “Null move and deep search: Selective-search heuristics for obtuse chess programs”, *ICCA Journal*, 16(3) (1993)137-143.

[4] Hyatt, R.M., “Rotated bitmaps, a new twist on an old idea”, *ICCA Journal,* 22 (4) (1999) 213-222.

[5] Rajlich, V., *Rybka* chess engines information could be found at: http://www.rybkachess.com 2005.

[6] Shannon, C.E., “Programming a computer for playing chess”, *Philosophical Magazine*, 41(7) (1988) 256-275. Reprinted in *Computer Games I* (ed. D.N.L. Levy) Springer-Verlag, New York, N.Y., (1950) 81-88.

[7] Slate, D.J. and Atkin, L.R., “CHESS 4.5 – The Northwestern University Chess Program”, *Chess Skill in Man and Machine,* P.W. Frey (ed.), Springer-Verlag, New York, N.Y. 2nd ed., 1983, 82-118.

[8] Vučković, V., and Vidanović, Đ., "An algorithm for the detection of move repetition without the use of hash-keys", *Yugoslav Journal of Operations Research (YUJOR*), 17(2) (2007) 257-274.

[9] Vučković, V. “The theoretical and practical application of the advanced chess algorithms”, PhD Theses, The Faculty of Electronic Engineering, The University of Nis, Serbia, 2006.

[10] Vučković, V., *Axon/Achilles* experimental chess engines information could be found at: http://chess.elfak.ni.ac.yu ,2001.

[11] Zobrist, A.L.,”A new hashing method with application for game playing”, Technical Report #88, Computer Science Department, The University of Wisconsin, Madison, WI, USA, 1970, reprinted in *ICCA Journal*, 13 (2) (1990)69-73.

[12] Vučković, V., "The compact chessboard representation", *ICGA Journal*, 31(3)(2008) 157-164.

[13] Vučković, V., "The method of the chess search algorithms parallelization using two-processor distributed system", *The Scientific Journal Facta Universitatis, Series Mathematics and Informatics*, 22 (2)(2007) 175-188.

[14] Šolak, R., and Vučković, V., "Time management during a chess game", *ICGA Journal*, 32(4)(2010) 206- 220.

[15] Vučković, V., and Šolak, R., "Time management procedure in computer chess", *Facta Universitatis Series: Automatic Control and Robotics*, 8 (1)(2009) 75-87.

[16] Vučković, V., *Napredni šahovski algorimi i sistemi*, Zadužbina Andrejević, Biblioteka Dissertatio, Beograd 2011.

[17] Vučković, V., “The Realization of the parallel chess system using UDP communication protocol”, *Proceedings of the VIII International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services TELSIKS 2007*, Niš, 26-28. September 2007, (2) 450-453.

[18] Vučković, V., “Realizacija efikasnog generatora poteza u šahovskim aplikacijama visokih performansi”, *Zbornik radova sa 52. Konferencije ETRAN-a, CD ROM Proceedings, Sekcija Veštačka inteligencija*, rad VI2.3, Palić, 8-12. juni 2008.

[19] Rajković, P., and Vučković, V., “Osnovni elementi heurističke evaluacione funkcije”, *Zbornik radova sa 52. Konferencije ETRAN-a,, CD ROM Proceedings, Sekcija Veštačka inteligencija,* rad VI2.4, Palić, 8-12. juni 2008.

[20] Vučković, V., “Specijalni elementi evaluacione funkcije”, *Zbornik radova sa 53. Konferencije ETRAN-a,, CD ROM Proceedings, Sekcija Veštačka inteligencija*, рад VI1.3, Vrnjačka Banja, 15 – 18. juna 2009.

[21] Vučković, V., Napredni null-move R=3 algoritam za obradu stabla u logičkim igrama, *Zbornik radova sa 54. Konferencije ETRAN-a,, CD ROM Proceedings, Sekcija algoritmi*, рад 7.1, Donji Milanovac, 7-11. Juna 2010.

[22] Vučković, V., and Kovačević, M., “*Achilles* – multimedijalna prezentacija”, Monografska publikacija u elektronskom izdanju, Elektronski fakultet u Nišu, Niš, 2008.