Clausthal University of Technology
Department of Informatics
Julius-Albert-Str. 4
D-38678 Clausthal-Zellerfeld

Master thesis

Finding Hash Functions for Bitboard Based Move Generation

Niklas Fiekas
Student number: 404068

27th April 2018

First reviewer: Prof. Dr. Jürgen Dix
Second reviewer: Prof. Dr. Thorsten Grosch

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß übernommen worden sind, sind kenntlich gemacht. Weiterhin versichere ich, dass diese Arbeit keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt worden ist.

Desweiteren erkläre ich mich damit einverstanden, dass meine Masterarbeit in der Instituts- und/oder Universitätsbibliothek ausgelegt und zur Einsichtnahme aufbewahrt werden darf.

Clausthal, 27.04.2018

Abstract

Most strong chess and checkers engines use 64-bit integers (bitboards) to represent piece positions on the 64 squares of the board. On modern processors many useful operations such as unions, intersections, shifts and counts can then be expressed with single CPU instructions. Attacked squares and possible move targets are usually looked up in a precomputed hash table.

We review state of the art methods for retrieving sliding piece attacks (PEXT bitboards, PEXT/PDEP bitboards, fixed-shift and fancy magic bitboards, black magics), all of which are based on perfect hash functions, and introduce lower and upper bounds for so-called magic factors. Combined with a fast algorithm for magic testing we can then find magic factors with desirable properties or disprove their existence by exhaustive enumeration.

Zusammenfassung

Die meisten starken Schachprogramme greifen auf 64-bit Werte (Bitboards) zurück, um Teilmengen von Feldern zu codieren. Moderne Prozessoren erlauben dann nützliche Operationen wie Verschiebungen und das Bilden von Vereinigungen und Schnittmengen mit einzelnen CPU-Instruktionen. Angegriffene Felder werden üblicherweise in Hashtabellen nachgeschlagen.

In der vorliegenden Arbeit werden bekannte Methoden dazu (Magics mit festem und variablem Shift, PEXT Bitboards, PEXT/PDEP Bitboards, Black Magics) aufgearbeitet. Der Hauptbeitrag sind Schranken für sogenannte magische Faktoren. Mit Hilfe eines schnellen Algorithmus zum Testen von magischen Faktoren können dann alle Kandidaten für bestimmte Felder überprüft werden. Damit kann zum ersten Mal bestimmt werden, ob magische Faktoren mit gewünschten Eigenschaften existieren.

Contents

List of Figures

1. Introduction

Search techniques and evaluation heuristics for chess-like games are widely studied in the domain of artificial intelligence. Other than these a fast implementation of the basic rules themselves is an important factor for the strength of a chess playing program. One particular challenge (and the focus of this thesis) is to efficiently determine squares reachable by long-range sliding moves based on a bitboard representation of piece positions.

In the last decade there have been several advances in this area: "Kindergarten bitboards" [26] replacing rotated bitboards with precomputed hash tables of possible piece configurations on diagonals, ranks and files, "Magic bitboards" [16, 25], also based on hash tables, one bishop and one rook table for each square, and finally "PEXT bitboards" [29] using the parallel bit extract instruction introduced with Intel's Haswell processors in 2013.

Today the state of the art is to use PEXT bitboards when a fast pext instruction is available (on Intel processors since Haswell) and magic bitboards everywhere else. Based on 100 runs of Stockfish's benchmark suite PEXT bitboards give a speedup of 2.3%[1] over magic bitboards.

Magic bitboards have not received much attention since, although an idea presented by Volker Annuss almost closes the gap. His simulated annealing algorithm exploits unused buckets in the hash tables of individual squares to create a single overlapping table that is 21% more compact [4]. Smaller tables are expected to improve CPU cache efficiency. We implemented and benchmarked a proof of concept in Stockfish[2] to measure this effect, and it reduces the advantage of PEXT bitboards to 0.7%.

It may be rewarding to analyze this more closely: Are further improvements to magic bitboards possible, perhaps even beating PEXT bitboards, or should we be satisfied with PEXT bitboards (where available) and shift our attention elsewhere? So far answering this question has not been computationally feasible. The main contribution of this thesis is to significantly reduce the search space for candidate hash functions and to speed up the testing of candidates. This allows us to find hash functions with proven optimal index ranges, of which some are given in the appendix and some will be computed in the months to come.

[1] Individual results within $\pm 0.1\%$. Base commit was b88374b14a7. All benchmarks in this thesis were produced on an Intel Core i7-6850K CPU @ 3.60GHz.

[2] `https://github.com/official-stockfish/Stockfish/pull/1538/files`

1.1. Related work

Hash functions are important in theory and practice, and associated data structures are used on almost any computer system. When the set of possible keys is known there are schemes to efficiently construct minimal perfect hash functions with guaranteed success, such as Hash and Displace [24, 5]. However in general encoding the minimal perfect hash function itself takes space linear in the number of keys. In chess programming we would rather trade such guarantees for performance at runtime, even if that means spending significant amounts of computational resources in advance.

Magic bitboards rely on a family of universal hash functions (in the sense of Carter and Wegman [7]) introduced by Dietzfelbinger et al. [10]. These can be evaluated with a single integer multiplication and bit shift. Universality offers some probabilistic guarantees for functions picked at random. In particular we should be able to find a hash function that maps a set of 2^c piece configurations on files, ranks or diagonals to an output of $w \geq 2c$ bits. Monge analyses multiplicative hash functions for inputs with sparse digit representations (which applies to bitboard representations of files, ranks, diagonals and their subsets) and improves this bound to $w \geq \log_2(3)c$ [20]. Monge's work is motivated by magic bitboards in chess programming (not the other way around) where hash functions with $w = c$ and even some with $w = c - 1$ are known, better than can be expected in general. We show that this bound is sharp for the particular inputs encountered for some squares.

Some alternatives to Dietzfelbinger's hash function can be considered. Fenner and Levene design minimal perfect hash functions specifically with sliding piece attacks in mind [11]. Unfortunately evaluating them requires more cycles than a single multiplication and shift.

Although rigorous statistical tests are used to ensure progress or absence of regressions, it should be noted that most advances in chess programming are discussed casually on mailing lists or in discussion forums, for example `fishcooking@googlegroups.com` and `http://talkchess.com`, rather than in scientific publications. We review the state of the art based on these discussions and publicly shared code in Chapter 3. Much in this style, Kannan explains magic multiplication in a "casual paper" [17]. He proposes a way to efficiently find appropriate magic factors. This is essentially using De Bruijn sequences [6] when bits of the possible inputs do not overlap. When they do overlap (as is the case for sliding piece attacks) he still has to resort to trial and error.

1.2. Structure of the thesis

This thesis is structured as follows. Chapter 2 introduces *bitboards* and reviews the efficient implementation of common set operations that warrant their prevalence in board game playing programs.

In Chapter 3 we comprehensively discuss various state of the art methods for retrieval of sliding piece attacks: PEXT bitboards, PEXT/PDEP bitboards, magic bitboards with variable and fixed shift and black magics. In doing this we establish common vocabulary, such as the terms *relevant occupancy* and *magic factor*.

The main contribution are the bounds for magic factors presented in Chapter 4, significantly reducing the search space from 2^{64} candidates to between 2^{26} and 2^{63} candidates (depending on piece type and square). Even then the search space is quite large. We therefore improve upon the state of the art method for testing individual candidates in Chapter 5.

All in all this allows us to exhaustively test candidate factors for some squares. Chapter 6 summarizes the current results and gives an outlook on future work. Some concrete magic factors with proven optimal index ranges are given in the appendix. The code used to compute these is published in a public repository `https://github.com/niklasf/magics`.

2. Bitboards

Bitboards are a natural and compact representation of sets of board positions (squares) [1]. Complements, unions, intersections and symmetrical difference map directly to bitwise operations. On modern CPUs with 64-bit registers these are single CPU instructions. This chapter discusses details and established fast implementations of other important set operations that are used throughout this thesis and that make bitboards a popular choice for chess and other board game engines.

Checker boards are symmetrical. The following sections use a canonical numbering $\mathbb{S} = \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{62}, \mathbf{63}\}$ for all 64 squares **a1**, **a2**, ..., **h7**, **h8** of the checker board as shown in Figure 2.1.

Definition 1. *A **bitboard** is a 64-bit integer representing a set of squares S.*

$$\mathrm{B}(S) = \sum_{\mathrm{s} \in S} 2^{\mathrm{s}}$$

For example the set of the four corner squares **a1**, **h1**, **a8**, **h8** would be expressed as 10000001 00000000 00000000 00000000 00000000 00000000 00000000 10000001 in binary notation or `0x8100000000000081` in hexadecimal notation. The spaces in the binary notation correspond to one group of 8 bits for each horizontal rank of the board. The starting positions of the white pawns **a2**, **b2**, ..., **g2**, **h2** would be `0x000000000000ff00` in hexadecimal notation. The set of all 32 light squares **b1**, **d1**, ..., **e8**, **g8** is `0x55aa55aa55aa55aa`.

2.1. Iterating squares

To iterate over the squares of a bitboard `b` in increasing order we repeatedly identify the minimum square, i.e. the index of the least significant bit `lsb(b)`, and then remove it from the set. Similarly a bitboard can be iterated in reverse order by repeatedly identifying and removing the maximum square, i.e. the most significant bit `msb(b)`.

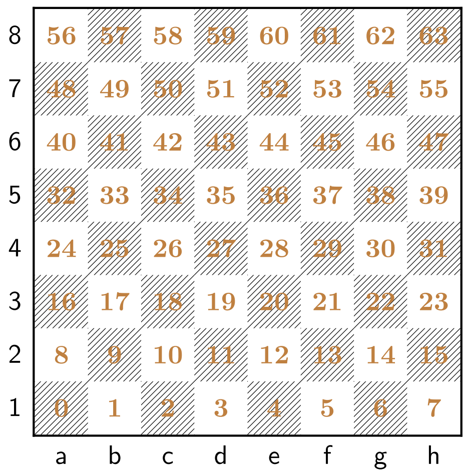


Figure 2.1.: Canonical square numbering

$\emptyset$	`UINT64_C(0)`	
$\{s\}$	`UINT64_C(1) << s`	shl
$\overline{A}$	`~A`	not
$A \cup B$	`A | B`	or
$A \cap B$	`A & B`	and
$A \oplus B$	`A ^ B`	xor
$\{\min(A)\}$	`A & -A`	
$A \setminus \{\min(A)\}$	`A & (A - 1)`	
$s \in A$	`(A & (UINT64_C(1) << s)) != 0`	

Figure 2.2.: Basic bitboard set operations in C code and corresponding x86 instructions

```
while (b) {
    int s = lsb(b);

    // Trick to unset the least significant bit:
    // Subtraction with carry ripples through any
    // leading 0 bits and unsets the first 1 bit.
    // Masking with the original bitboard restores
    // leading zeros.
    b &= b - 1;
}
```

We need to find a fast implementation for `lsb` and `msb`.

$$\text{lsb}(\text{B}(S)) = \min(S)$$

$$\text{msb}(\text{B}(S)) = \max(S)$$

The zero-based index of the least significant bit is the number of trailing zeros in the binary representation. The index of the most significant bit is 63 minus the number of leading zeros. This allows using the GCC compiler intrinsics `__builtin_ctzll` and `__builtin_clzll` which map to bsfq and bsrq on x86 [12].

```
int lsb(uint64_t b) {
    return __builtin_ctzll(b);
}

int msb(uint64_t b) {
    return 63 - __builtin_clzll(b);
}
```

Software fallbacks can be implemented using De Bruijn multiplication [6]. The following variant was proposed by Kim Walisch [28]:

```
const int LSB_TABLE[64] = {
     0, 47,  1, 56, 48, 27,  2, 60,
    57, 49, 41, 37, 28, 16,  3, 61,
    54, 58, 35, 52, 50, 42, 21, 44,
    38, 32, 29, 23, 17, 11,  4, 62,
    46, 55, 26, 59, 40, 36, 15, 53,
    34, 51, 20, 43, 31, 22, 10, 45,
    25, 39, 14, 33, 19, 30,  9, 24,
    13, 18,  8, 12,  7,  6,  5, 63
};
```

```
int lsb(uint64_t b) {
    const uint64_t debruijn64 = UINT64_C(0x03f79d71b4cb0a89);
    return LSB_TABLE[((b ^ (b - 1)) * debruijn64) >> 58];
}
```

Finding the most significant bit $\lfloor \log_2 b \rfloor$ is susceptible to a divide and conquer approach as introduced by Eugene Nalimov [21]:

```
int msb(uint64_t b) {
    int s = 0;

    if (b > 0xffffffff) {
        b >>= 32;
        s = 32;
    }

    if (b > 0xffff) {
        b >>= 16;
        s += 16;
    }

    if (b > 0xff) {
        b >>= 8;
        s += 8;
    }

    // We could continue dividing, but it is
    // more efficient to look up the most
    // significant bit of the final byte in
    // a small table.
    return s + MSB_TABLE[b];
}
```

2.2. Counting squares

In some cases, for instance counting material or evaluating piece mobility, it is sufficient to count the squares in a bitboard, rather than enumerating each one.

$$\mathrm{popcount}(\mathrm{B}(S)) = |S|$$

Population counts are also frequently used outside of chess programming to compute the Hamming weight or Hamming distance of bit vectors. Once again GCC offers a compiler intrinsic `__builtin_popcountll` that maps directly to a CPU instruction if available [12]. There is a popcnt instruction on x86, popcount and popcount7 on arm.

```
int popcount(uint64_t b) {
    return __builtin_popcountll(b);
}
```

A simple and fast software fallback implementation is to divide the 64-bit integer into groups of 16 or 8 bits and look up their population counts in a precomputed table.

2.3. Iterating subsets

The following method to iterate over all $2^{|\mathrm{B}^{-1}(b)|}$ subsets of a bitboard b was coined "Carry-Rippler" by Marcel van Kervinck [18]. When iterating subsets of the whole board it is equivalent to incrementing the subset bitboard by 1 for each iteration. If there are some squares that are not present in b the subtraction with carry "ripples" through them. This is similar to the trick `b &= b - 1` from Section 2.1.

```
uint64_t subset = 0;
do {
    // Use subset here.
    subset = (subset - b) & b;
} while(subset);
```

2.4. Stepping piece attacks

Knights, kings and pawns in chess and men in draughts only attack (i.e. threaten to capture) a single square in each allowed direction as depicted in Figure 2.3. They are called stepping pieces or steppers in contrast to sliding pieces or sliders.

The directions can be given as deltas in the square numbering scheme. The deltas for attacks by a white pawn are $\{7, 9\}$, i.e. north-west and north-east, and $\{-7, -9\}$ for a black pawn. Similarly the deltas for king attacks are $\{8, 9, 1, -7, -8, -9, -1, 7\}$ clockwise from north. Knight deltas are $\{17, 10, -6, -15, -17, -10, 6, 15\}$. Using these, a relatively slow routine to generate attack bitboards can be implemented.

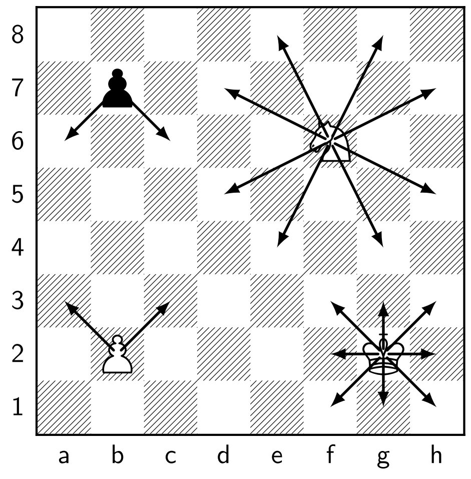


Figure 2.3.: Stepping piece attacks in chess

```
uint64_t stepper_attacks(int square, int deltas[]) {
    uint64_t attacks = 0;

    for (int i = 0; deltas[i]; i++) {
        int sq = square + deltas[i];

        // Do not step over the edge of the board.
        if (sq < 0 || 64 <= sq ||
            square_distance(sq, sq - deltas[i]) > 2) {
            continue;
        }

        attacks |= UINT64_C(1) << sq;
    }

    return attacks;
}
```

But there are only 64 distinct attack sets for each piece type, one for each square! So these can easily be precomputed and stored in a lookup table. It should be noted that the table size can be reduced by exploiting symmetries, but adding extra instructions to do that slows down the look-ups. All remaining sections are concerned with finding a way to use fast lookup tables also for sliding piece attacks.

3. Sliding piece attacks

This chapter introduces the terms *relevant occupancy* and *magic factor* and reviews state of the art methods for retrieving attack sets of sliding pieces.

Bishops, rooks and queens in chess are sliding pieces. They can move any number of squares in the allowed directions, *but cannot step over pieces that are blocking the path.* A simple but slow method is implementing sliding attacks as a function of the square, deltas for the directions and a bitboard containing the squares occupied by blocking pieces.

```c
uint64_t slider_attacks(int square, int deltas[],
                        uint64_t occupied) {
    uint64_t attacks = 0;

    for (int i = 0; deltas[i]; i++) {
        int sq = square;

        do {
            sq += deltas[i];

            // Do not slide over the edge of the board.
            if (sq < 0 || 64 <= sq ||
                square_distance(sq, sq - deltas[i]) > 1) {
                break;
            }

            attacks |= UINT64_C(1) << sq;

            // Stop after hitting a blocking piece.
        } while (!(occupied & (UINT64_C(1) << sq)));
    }

    return attacks;
}
```

We can concentrate on rook and bishop attacks because queens attacks are simply the union of rook attacks and bishop attacks for a given square.

```
uint64_t queen_attacks(int square, uint64_t occupied) {
    return (rook_attacks(square, occupied) |
            bishop_attacks(square, occupied));
}
```

Nonetheless, creating a lookup table with 2^{64} (or even only about $\frac{64!}{32! \cdot 32!} \approx 2^{60.7}$) entries for the possible occupancies is not feasible. We can substantially reduce this number by observing that for a given piece position many squares cannot influence the outcome. Only squares that would be attacked by the piece on an empty board are potential blocking squares. Furthermore the corner squares can never block piece attacks, because there is no square behind them. Similarly the edges of the board are only relevant if the piece is already on that edge.

Definition 2. *The sets of* ***relevant occupancies*** *can be defined as:*

$$R_{rook}(\mathbf{s}) = \{S \subseteq \mathbb{S} \,|\, \forall \mathbf{a} \in S : \mathit{rook\_attacks}(\mathbf{s}, \mathrm{B}(\{\mathbf{a}\}) \neq \mathit{rook\_attacks}(\mathbf{s}, \mathrm{B}(\emptyset))\}$$

$$R_{bishop}(\mathbf{s}) = \{S \subseteq \mathbb{S} \,|\, \forall \mathbf{a} \in S : \mathit{bishop\_attacks}(\mathbf{s}, \mathrm{B}(\{\mathbf{a}\}) \neq \mathit{bishop\_attacks}(\mathbf{s}, \mathrm{B}(\emptyset))\}$$

The empty set is always a relevant occupancy. (There can always be no blocking pieces whatsoever).

$$\min_{R \in R_{\mathrm{rook}}(\mathbf{s})} \mathrm{B}(R) = 0$$

$$\min_{R \in R_{\mathrm{bishop}}(\mathbf{s})} \mathrm{B}(R) = 0$$

All subsets of a relevant occupancy are also relevant occupancies. So we can use the largest relevant occupancy as a mask with bitwise-and to remove irrelevant squares from a bitboard of potential blocking pieces.

$$\max_{R \in R_{\mathrm{rook}}(\mathbf{s})} B(R) = \mathrm{B}\left(\bigcup R_{\mathrm{rook}}(\mathbf{s})\right)$$

$$\max_{R \in R_{\mathrm{bishop}}(\mathbf{s})} B(R) = \mathrm{B}\left(\bigcup R_{\mathrm{bishop}}(\mathbf{s})\right)$$

For example Figure 3.1 highlights the maximum relevant occupancy $\max_{R \in R_{\mathrm{rook}}(\mathbf{d6})} \mathrm{B}(R)$ in green. It illustrates that the pawn on **a6** and the queen on **f4** are not relevant to the rook attack set. They can be removed by bitwise-and with the green mask. Only the bishop on **d3** is actually blocking an attack.

Counting the number of relevant occupancies, there are 10 relevant squares if the rook is on one of the 36 non-edge squares, as can be seen in the previous example. Figure 3.2 shows there are 12 relevant squares for a rook in one of the 4 corners. 11 squares are relevant if the rook is on one of the remaining squares on the edge of the board, as shown in Figure 3.3.

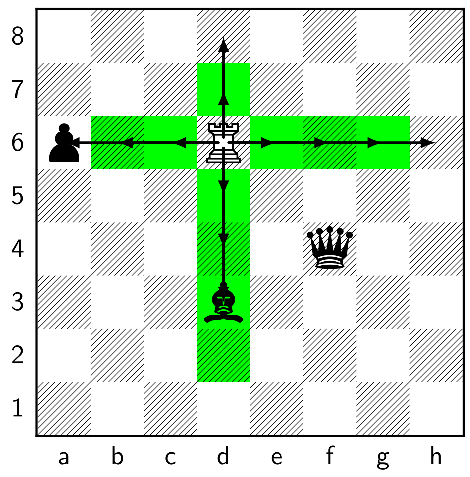


Figure 3.1.: Relevant blocking squares for rooks attacks from **d6**

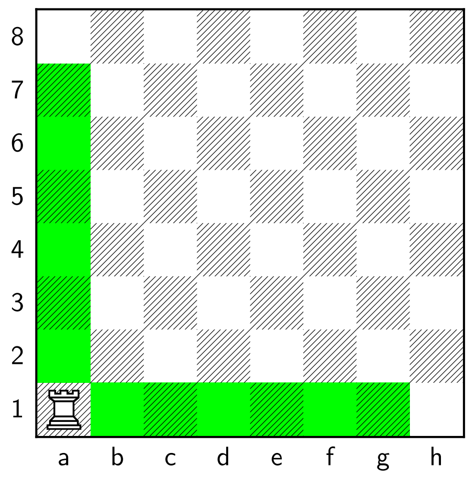


Figure 3.2.: Relevant blocking squares for rook attacks from **a1**

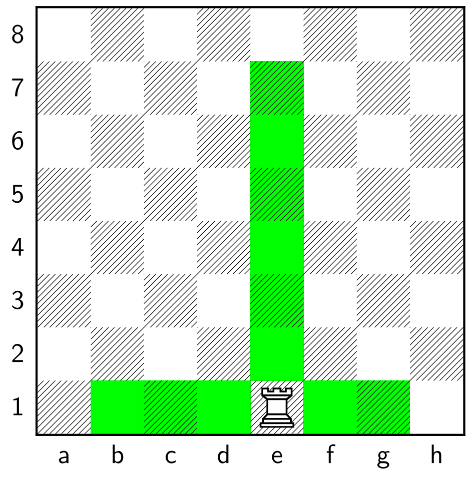


Figure 3.3.: Relevant blocking squares for rook attacks from **e1**

Hence:

$$\sum_{s \in \mathbb{S}} |R_{\text{rook}}(s)| = 4 \cdot 2^{12} + 24 \cdot 2^{11} + 36 \cdot 2^{10} = 102400$$

Creating a hash table with 102400 entries is feasible. Figure 3.4 shows that bishops only have between 2^5 and 2^9 relevant occupancies, so a lookup table for bishop attacks is feasible as well.

$$\sum_{s \in \mathbb{S}} |R_{\text{bishop}}(s)| = 4 \cdot 2^{9} + 12 \cdot 2^{7} + 4 \cdot 2^{6} + 44 \cdot 2^{5} = 5248$$

It remains to implement these hash tables in a way that supports very fast lookup. In particular we would like to avoid hash collisions that require branching. The following sections discuss and compare state of the art approaches. Chapters 4 and 5 are new contributions to this problem.

3.1. PEXT bitboards

To build the lookup table for sliding piece attacks we need hash functions, one for each square and piece type, that map relevant occupancies to a set of integer indexes.

$$h_{\text{rook},s} : R_{\text{rook}}(s) \to \mathbb{N}_0$$

$$h_{\text{bishop},s} : R_{\text{bishop}}(s) \to \mathbb{N}_0$$

Definition 3. *A hash function h: $A \to \mathbb{N}_0$ is called **perfect hash function** if it is injective, i.e. no two elements of A collide [14].*

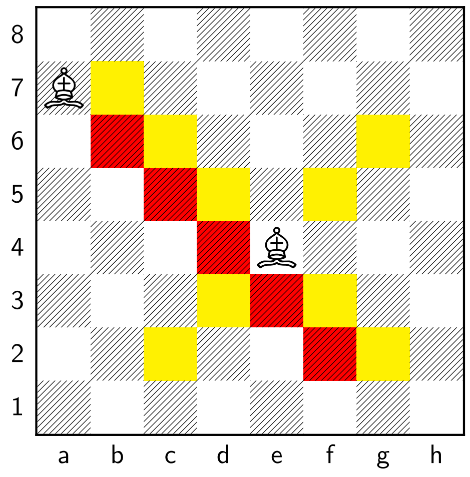


Figure 3.4.: Relevant blocking squares for bishop attacks from **a1** and **e4**

Using a perfect hash function avoids branching on collisions and achieves constant worst-case time for look-ups rather than just amortized constant time.

Definition 4. *A perfect hash function $A \to \{0, \ldots, |A| - 1\}$ is called **minimal perfect hash function**, i.e. it maps all keys to consecutive integers without gaps [5].*

Having few unused gaps (or none) in the lookup table is advantageous for CPU cache efficiency.

One family of minimal perfect hash functions is given by the *parallel bit extract* instruction pext on x86 [29]. It takes a 64-bit integer (in this case the bitboard of occupied squares) and extracts only the bits selected by a 64-bit mask (here the maximum relevant occupancy, drawn green in Figure 3.5).

$$h_{\mathrm{rook},\mathbf{s}}(A) = \operatorname{pext}\left(\mathrm{B}(A), \max_{R \in R_{\mathrm{rook}}(\mathbf{s})} \mathrm{B}(R)\right)$$

$$h_{\mathrm{bishop},\mathbf{s}}(A) = \operatorname{pext}\left(\mathrm{B}(A), \max_{R \in R_{\mathrm{bishop}}(\mathbf{s})} \mathrm{B}(R)\right)$$

pext is available via the `_pext_u64` intrinsic from `immintrin.h` [13]. A relatively slow but equivalent software fallback is:

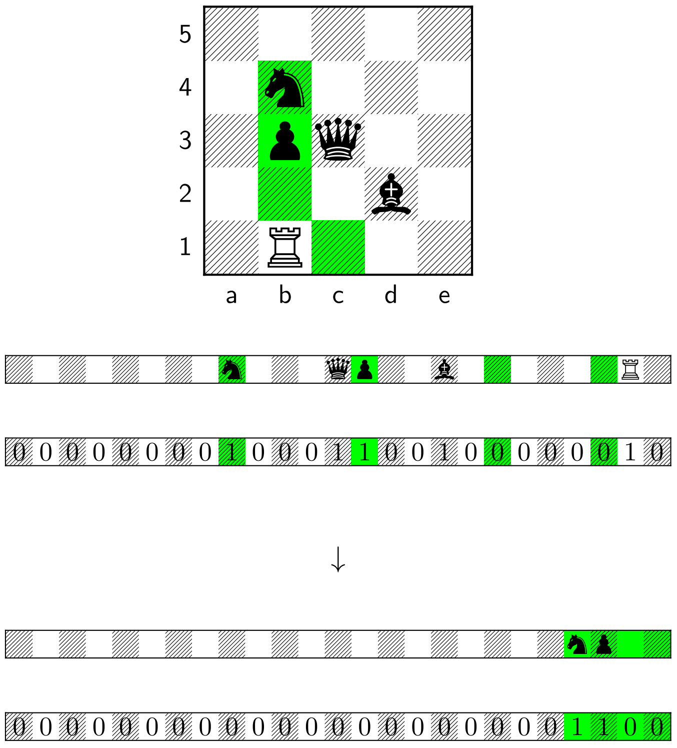


Figure 3.5.: PEXT on a 5x5 board

```
uint64_t pext(uint64_t b, uint64_t mask) {
    uint64_t result = 0;
    for (uint64_t bit = 1; mask; bit <<= 1) {
        if (b & mask & -mask) result |= bit;
        mask &= mask - 1;
    }
    return result;
}
```

Availability of pext immediately disqualifies other more general approaches to construct minimal perfect hash functions, such as Hash and Displace. These require complementary tables of at least the same order of magnitude as the hash table itself to encode the hash function [5].

As of April 2014, Stockfish, champion of TCEC Season 9, uses PEXT bitboards when enabled [9]. However pext is part of the bmi2 instruction set. Useful parallel bit extract is only available on Intel processors since Haswell. Previous processor generations have no support. AMD implements only a slow pext, and arm does not support pext whatsoever.

Also, albeit being "minimal", the attack tables are still quite large $((102400 + 5248) \cdot 64\text{bit} = 861.2\text{kB})$. The approaches in the following sections use more insights into the problem, alter it slightly and achieve smaller table sizes, trading extra instructions for cache efficiency.

3.2. PEXT/PDEP bitboards

PEXT bitboards store 64-bit attack sets in the hash table. These can be compressed with an approach that resembles the idea behind relevant occupancies, i.e. a rook or bishop on a given square can only attack subsets of at most 14 squares. This means a rook attack set can be compressed with pext and stored in a 14-bit (but more practically 16-bit) integer [29].

$$v = \text{compress}_{\text{rook}}(\text{s}, A) = \text{pext}(\text{B}(A), \text{B}(\text{rook\_attacks}(\text{s}, \emptyset)))$$

pext is reversible, and indeed there is a CPU instruction pdep on x86 for parallel bit deposit, available via the intrinsic `_pdep_u64` from `immintrin.h` [13].

$$\text{B}(A) = \text{decompress}_{\text{rook}}(\text{s}, v) = \text{pdep}(v, \text{B}(\text{rook\_attacks}(\text{s}, \emptyset)))$$

The following software fallback is given for completeness, but generally too slow for practical use.

```
uint64_t pdep(uint64_t v, uint64_t mask) {
    uint64_t result = 0;
    for (uint64_t bit = 1; mask; bit <<= 1) {
        if (v & bit) result |= mask & -mask;
        mask &= mask - 1;
    }
    return result;
}
```

The total table size is $(102400 + 5248) \cdot 16\text{bit} = 215.3\text{kB}$. Like PEXT itself, PDEP is only useful on recent Intel CPUs. Whether the extra instruction is worth the gained cache efficiency depends on the application. PDEP is not currently used in Stockfish.

3.3. Magic bitboards

This section shows how a family of multiplicative hash functions introduced by Dietzfelbinger et al. [10] can be used as a serious alternative to PEXT. In the presented form it was used in chess programs since 2006 [16, 25], before PEXT became available with Intel's Haswell microarchitecture in 2013. Unlike PEXT, fast integer multiplication is widely available across CPU vendors, and even on the latest Intel processors magic bitboards are only narrowly outperformed (0.7% speedup in Stockfish, and the main contributions of this thesis offer the prospect to close that gap).

A family $H = \{h : A \to \{0, \dots, m-1\}\}$ of hash functions is *universal* if any two elements are unlikely to collide under a random candidate function [7]:

$$\forall x, y \in A, x \neq y : \Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}$$

Dietzfelbinger gives such a family, here instantiated with a register width of 64 bit:

$$H_w = \{h_{\alpha,w} : \{0, \dots 2^{64} - 1\} \to \{0, \dots, 2^w - 1\} \mid \alpha \text{ odd}\}$$
$$h_{\alpha,w}(x) = \alpha \cdot x \bmod 2^{64} \operatorname{div} 2^{64-w}$$

Note that these functions can be very efficiently implemented as `(a * x) >> (64 - w)`. Integer division and modular arithmetic merely amount to a shift and register overflow respectively.

The parameter w determines the width of the output in bits as depicted in Figure 3.6. Smaller w lead to smaller table sizes, but naturally also to higher chances of collisions. We are looking for table sizes competitive with PEXT, i.e. with w no larger than the number of squares in the maximum relevant occupancy. According only to the guarantees

$$\Pr_{h_{\alpha,w} \in H_w}[h_{\alpha,w} \text{ is bijective on } A] \geq 1 - \frac{|A|^2}{2^w}$$

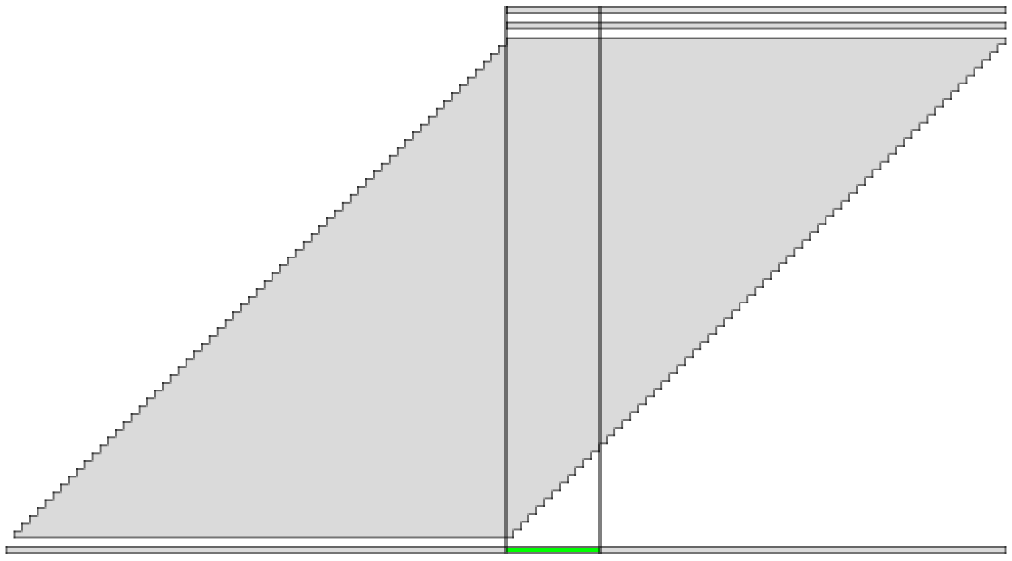


Figure 3.6.: Parallelogram shape of the 64-bit multiplication with bits selected as the hash value in green [22]

offered by universality [10] we cannot generally expect to find minimal perfect hash functions $h_{\alpha,w}$ on the set of relevant occupancies. However we do not care about colliding relevant occupancies that map to the same attack set. There is a total of $102400 + 5248$ relevant rook and bishop occupancies for all the squares, but only a total of $4900 + 1428$ distinct attack sets! Such *constructive collisions* may even help to achieve much smaller table sizes.

In the context of chess programming suitable factors α are called *magics* or *magic factors*. The technique of using these factors for bitboard based move generation is called *magic bitboards*. More precisely (and analogously for bishop squares):

Definition 5. *α is a* ***magic factor*** *for rook square s with shift w if all collisions are constructive:*

$$\forall x, y \in R_{rook}(\mathrm{s}) : h_{\alpha,w}(\mathrm{B}(x)) = h_{\alpha,w}(\mathrm{B}(y)) \implies \mathit{rook\_attacks}(\mathrm{s}, x) = \mathit{rook\_attacks}(\mathrm{s}, y)$$

Observe that a magic factor with shift w is also always a magic factor with shift $w+1$. Adding superfluous bits cannot introduce new collisions. However it is likely to increase the index range of the square and the table size. We are therefore interested in finding magics with minimal shift and minimal index range.

Pending the results from the following chapters, brute force testing of random candidates is the best known algorithm for finding magic factors. For each square s rook magics with shift

$$w = \operatorname{popcount}\left(\max_{R \in R_{\mathrm{rook}}(\mathrm{s})} B(R)\right)$$

and bishop magics with shift

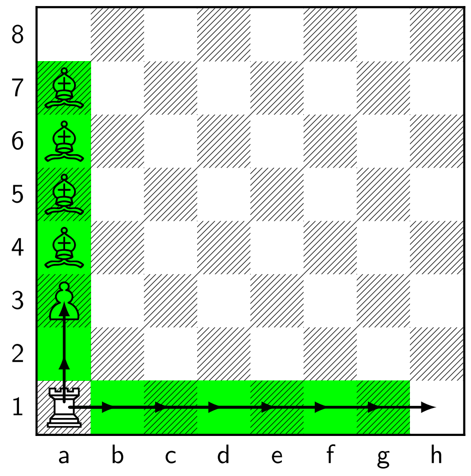


Figure 3.7.: A rook on **a1** has $2^{12} = 4096$ relevant occupancies, but only $7 \cdot 7 = 49$ distinct attack sets. In this example positions with or without any of the bishops can collide constructively.

$$w = \text{popcount}\left(\max_{R \in R_{\text{bishop}}(\mathbf{s})} B(R)\right)$$

can be found in milliseconds. Stockfish even generates them every time it starts [8]. This gives hash functions on a par with a minimal perfect hash function on the relevant occupancies. For some squares intensive search yielded improved magics [2] with:

$$w = \text{popcount}\left(\max_{R \in R_{\text{rook}}(\mathbf{s})} B(R)\right) - 1$$

But generally testing all 64-bit integer candidates is not feasible, so the best magics for most squares remain unknown.

A typical magic bitboard implementation would store the magic factor, maximum relevant occupancy and shift for each square.

```
typedef struct {
    uint64_t factor; // a
    uint64_t mask;   // maximum relevant occupancy
    int shift;       // 64 − w
    int offset;      // offset into the attack table
} magic_t;

// constants for each square computed in advance
magic_t square_magics[64] = { ... };

// hash table initialized at startup
uint64_t attack_table[107648];

uint64_t rook_attacks(int square, uint64_t occupied) {
    magic_t m = square_magics[square];
    unsigned index = ((occupied & m.mask) * m.factor) >> m.shift;
    return attack_table[m.offset + index];
}
```

3.4. Fixed shift magics

This can be tweaked in many ways. A promising idea reintroduced by Volker Annuss is to use the same shift w (the maximum shift over all squares) as a "common denominator" for all squares, so that it can be hard-coded and does not have to be fetched from the table. He then proposes to find magics that either produce small index ranges regardless of the suboptimal shifts or to use the resulting gaps in the table for other squares. Packing hash tables with gaps seems to be a hard optimization problem, but in practice good results can be achieved with simulated annealing. The smallest known table size is $88772 \cdot 64\text{bit} = 710.2\text{kB}$ [4].

The search for such magics also benefits from the techniques presented in this thesis. It would be particularly important to find improved rook magics for the four corner squares (so far only **a8** or **h8** are known). These rook squares have the maximum shift 12 which can potentially be reduced to 11, effectively almost cutting the table size in half (but not quite, since the space gained by packing need not improve proportionally).

Similarly only the four center squares **d4**, **e4**, **d5**, **e5** with shift 9 prevent us from using fixed shift 8 for all bishop magics. We apply the results from Chapters 4 and 5 to find an optimal magic for **d5**, thereby showing that the maximum shift will remain 9.

3.5. Black magics

The key concept of PEXT bitboards and magic bitboards is to use a hash function on occupied squares, after reducing the key space by masking with relevant occupancies.

```
occupied & m.mask
```

This takes the value 0 for an empty occupancy, which is always mapped to 0 by a multiplicative hash function. With the lowest index fixed to 0, the index range may only be reduced from the top. An alternative idea introduced by Volker Annuss [3] is to use:

```
occupied | ~m.mask
```

This reduces the key space by setting the irrelevant bits to 1 instead of zeroing them. Since this does not take the value 0 the index range may be reduced from the bottom as well as from the top. So far the tables produced with this approach are slightly smaller (708.1kB) but do not currently outperform the fixed shift tables. 519 further entries could be saved with the fast testing from Chapter 5.

From another perspective this is equivalent to adding the constant `~m.mask` to the masked occupancy.

```
occupied & m.mask + ~m.mask
```

It should be noted that this is really different from traditional (white) magics. The same result cannot necessarily be achieved by multiplying with a different magic factor. Unfortunately this also makes it harder to analyze the hash function analytically, so that the main result from Chapter 4 does not apply to black magics.

3.6. Incorporating extra information in magic factors

Since we pick one of many possible magic factors for each square we can use this choice to encode additional information. This is more cache efficient than using a separate table.

Grant Osborne proposed to pick magics so that the upper 6 bits can be used as the shift [23]. Similarly, Eugene Kotlov suggested using the lower 16 bits as the offset into the combined attack table [19].

These approaches also benefit from the results of this thesis. Chapter 4 gives insights into the bit structure of magics. Specifically we can always pick some upper bits arbitrarily. Furthermore, with the fast testing from Chapter 5 we can check if a magic factor with any particular bit pattern exists.

4. Bounds on magics

In this chapter we analyze the family of multiplicative hash functions to significantly reduce the search space for magics.

Taking the opposite approach serves as motivation. View the hash functions as a black box with no structure to analyze, i.e. just a random function $h : R_{\text{rook}}(\mathtt{s}) \to \{0, \dots, 2^w - 1\}$. Now we start assigning relevant occupancies to the $n_0 = 2^w$ buckets. Do this in groups of sizes $m_0, \dots, m_i, \dots$ that each map to the same attack set, so that they can collide constructively. After assigning the first m_0 relevant occupancies we expect

$$n_0 \cdot \left(1 - \left(\frac{n_0 - 1}{n_0}\right)^{m_0}\right)$$

buckets to be occupied. To do this iteratively let $u_0(k)$ be the probability that exactly k buckets are used after the first iteration. With second order Stirling numbers [15] in brace notation:

$$u_0(k) = \left\{ {m_0 \atop k} \right\} \frac{n_0!}{n^{m_0}(n_0 - k)!}$$

And with each iteration:

$$u_i(k) = \sum_{j=0}^{n_0} \sum_{\substack{l \text{ s.t.} \\ k=j+l}} u_{i-1}(j) \left\{ {m_i \atop l} \right\} \frac{(n_0 - j)!}{(n_0 - j)^{m_i}(n_0 - j - l)!}$$

This seems to be difficult to evaluate exactly, but an approximation yields that it should be exceptionally unlikely to find any improved rook magics. Yet 14 have been found with brute force search. Furthermore Figure 4.1 shows the difficulty of finding magics (anti-proportional to their density) does not seem to be fully symmetrical on the checker board as predicted by a black box model. Clearly there must be some structure!

4.1. Lower bound

We derive a simple lower bound for magics by showing that smaller candidates will inevitably let the two lowest relevant occupancies collide.

Proposition 1. *Let r_{max} be the maximum relevant occupancy of a square $\mathtt{s}$, α a magic factor with shift w for square $\mathtt{s}$. Then:*

$$2^{64 - w - \text{lsb}(r_{max})} \le \alpha$$

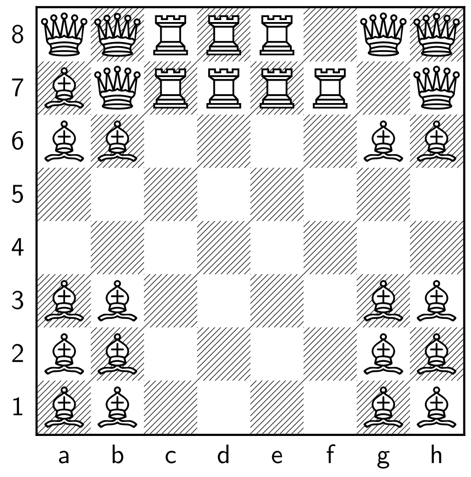


Figure 4.1.: Squares with previously known improved rook and/or bishop magics

Proof. Consider the two lowest relevant occupancies r_0, r_1. We know that $r_0 = 0$, i.e. it corresponds to a piece roaming freely, and that r_0 is always mapped to 0 by the multiplicative hash function. The second lowest relevant occupancy $r_1 \neq 0$ corresponds to exactly one blocked square $\text{lsb}(r_{\text{max}})$. Therefore the attack sets of r_0 and r_1 are distinct and they cannot collide constructively.

$$h_{\alpha,w}(r_1) = (r_1 \cdot \alpha \bmod 2^{64}) \operatorname{div} 2^{64-w} \neq h_{\alpha,w}(x_0) = 0$$

Integer division by 2^{64-w} selects the w uppermost bits of the 64-bit product. So α has to be sufficiently large, to ensure at least one of the uppermost w bits is set and the hash is non-zero. We get

$$2^{64-w} \leq r_1 \cdot \alpha \bmod 2^{64} = 2^{\text{lsb}(r_{\text{max}})} \cdot \alpha \bmod 2^{64}$$

or as a necessity in terms of α:

$$2^{64-w-\text{lsb}(r_{\text{max}})} \leq \alpha$$

□

For bishops the squares **a3**, **c1** and **g7** benefit the most with $\alpha \geq 2^{49}$, for rooks the squares **c1**, …, **g1** benefit the most with $\alpha \geq 2^{51}$ when looking for magics with improved shifts. This bound barely makes a dent in the search space of size 2^{64}, especially compared to the result in the following section. However it helps to avoid counter-productive reorderings of relevant occupancies with the fast testing method described in Chapter 5.

	a	b	c	d	e	f	g	h
8	2^{43}	2^{43}	2^{42}	2^{41}	2^{40}	2^{39}	2^{38}	2^{36}
7	2^{44}	2^{44}	2^{43}	2^{42}	2^{41}	2^{40}	2^{39}	2^{37}
6	2^{44}	2^{44}	2^{43}	2^{42}	2^{41}	2^{40}	2^{39}	2^{37}
5	2^{44}	2^{44}	2^{43}	2^{42}	2^{41}	2^{40}	2^{39}	2^{37}
4	2^{44}	2^{44}	2^{43}	2^{42}	2^{41}	2^{40}	2^{39}	2^{37}
3	2^{44}	2^{44}	2^{43}	2^{42}	2^{41}	2^{40}	2^{39}	2^{37}
2	2^{43}	2^{43}	2^{44}	2^{44}	2^{44}	2^{44}	2^{44}	2^{43}
1	2^{50}	2^{50}	2^{51}	2^{51}	2^{51}	2^{51}	2^{51}	2^{50}

Figure 4.2.: Lower bounds for rook magics with improved shift

	a	b	c	d	e	f	g	h
8	2^{43}	2^{36}	2^{28}	2^{20}	2^{25}	2^{33}	2^{41}	2^{48}
7	2^{45}	2^{44}	2^{36}	2^{28}	2^{33}	2^{41}	2^{49}	2^{48}
6	2^{46}	2^{45}	2^{42}	2^{34}	2^{39}	2^{47}	2^{48}	2^{47}
5	2^{47}	2^{46}	2^{43}	2^{40}	2^{45}	2^{46}	2^{47}	2^{46}
4	2^{48}	2^{47}	2^{44}	2^{45}	2^{44}	2^{45}	2^{46}	2^{45}
3	2^{49}	2^{48}	2^{47}	2^{46}	2^{45}	2^{44}	2^{45}	2^{44}
2	2^{41}	2^{40}	2^{41}	2^{40}	2^{39}	2^{38}	2^{37}	2^{36}
1	2^{48}	2^{48}	2^{49}	2^{48}	2^{47}	2^{46}	2^{45}	2^{43}

Figure 4.3.: Lower bounds for bishop magics with improved shift

4.2. Upper bound

In this section we look closer at the multiplication of relevant occupancies with the magic factor that is the heart of the hash function. We show that all magic candidates from $0, \ldots, 2^{64}-1$ are equivalent to a candidate in the much smaller range $0, \ldots, P(\mathbf{s})-1$, effectively introducing an upper bound for exhaustive search.

Proposition 2. *Let r_{max} be the maximum relevant occupancy of a square, α a magic factor with shift w for square* $\mathbf{s}$*.*

$$P(\mathbf{s}) = 2^{64-\operatorname{lsb}(r_{max})}$$

Then $\alpha + P(\mathbf{s}) \bmod 2^{64}$ is also a magic factor with shift w.

Proof. Let $0 < r_1 < \ldots < r_i < \ldots < r_{\max}$ be relevant occupancies for the square $\mathbf{s}$. Then for each r_i there is a smallest period $p(r_i) > 0$ such that:

$$r_i \cdot p(r_i) = 0 \pmod{2^{64}}$$

Multiplying with a larger factor is not useful. It will just wrap around (the factor $p(r_i)+1$ is equivalent to 1 in this sense). Since

$$r_i \cdot p(r_i) = \operatorname{lcm}(2^{64}, r_i) = \frac{2^{64} r_i}{\gcd(2^{64}, r_i)}$$

we get that:

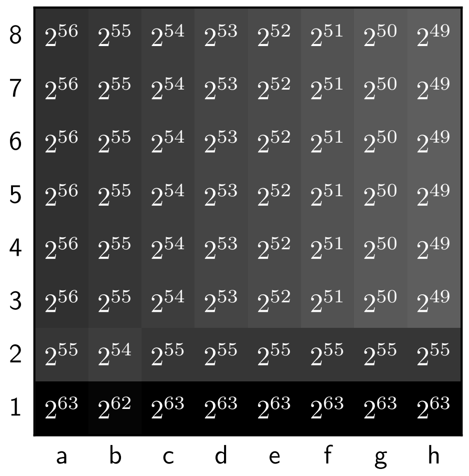
$$p(r_i) = \frac{2^{64}}{\gcd(2^{64}, r_i)} = 2^{64-\operatorname{lsb}(r_i)}$$

Now take the least common multiple of the $p(r_i)$ for all relevant occupancies of the square $\mathbf{s}$.

$$P(\mathbf{s}) = \operatorname{lcm}(p(r_1), \ldots, p(r_{\max})) = 2^{64-\operatorname{lsb}(r_{\max})}$$

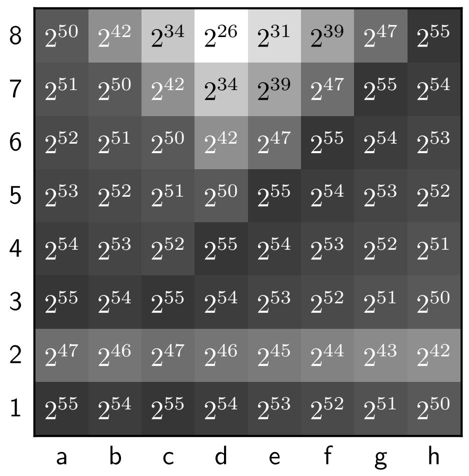
We know that $r_i \cdot P(\mathbf{s}) = 0 \pmod{2^{64}}$ for all relevant occupancies r_i. The proposition follows. In other words all magic candidates are equivalent to a candidate in the range $0, \ldots, P(\mathbf{s})-1$. $\square$

Among the bishop squares **d8** benefits the most with $P(\mathbf{59}) = 2^{26}$, **h8** benefits the least with $P(\mathbf{63}) = 2^{55}$. Rook squares generally remain more difficult, because they always have at least one relevant occupancy on the first or second rank. The squares **h3**, …, **h8** benefit the most with $P(\mathbf{23}) = \ldots = P(\mathbf{63}) = 2^{49}$. The squares **a1** and **c1**, …, **h1** benefit the least with $P(\mathbf{0}) = P(\mathbf{2}) = P(\mathbf{3}) = P(\mathbf{4}) = P(\mathbf{5}) = P(\mathbf{6}) = P(\mathbf{7}) = 2^{63}$, merely halving the search space.



	a	b	c	d	e	f	g	h
8	2^{56}	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}	2^{49}
7	2^{56}	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}	2^{49}
6	2^{56}	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}	2^{49}
5	2^{56}	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}	2^{49}
4	2^{56}	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}	2^{49}
3	2^{56}	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}	2^{49}
2	2^{55}	2^{54}	2^{55}	2^{55}	2^{55}	2^{55}	2^{55}	2^{55}
1	2^{63}	2^{62}	2^{63}	2^{63}	2^{63}	2^{63}	2^{63}	2^{63}

Figure 4.4.: Magic periods $P_{\text{rook}}(\mathbf{s})$ for rook squares



	a	b	c	d	e	f	g	h
8	2^{50}	2^{42}	2^{34}	2^{26}	2^{31}	2^{39}	2^{47}	2^{55}
7	2^{51}	2^{50}	2^{42}	2^{34}	2^{39}	2^{47}	2^{55}	2^{54}
6	2^{52}	2^{51}	2^{50}	2^{42}	2^{47}	2^{55}	2^{54}	2^{53}
5	2^{53}	2^{52}	2^{51}	2^{50}	2^{55}	2^{54}	2^{53}	2^{52}
4	2^{54}	2^{53}	2^{52}	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}
3	2^{55}	2^{54}	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}
2	2^{47}	2^{46}	2^{47}	2^{46}	2^{45}	2^{44}	2^{43}	2^{42}
1	2^{55}	2^{54}	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}

Figure 4.5.: Magic periods $P_{\text{bishop}}(\mathbf{s})$ for bishop squares

5. Fast magic testing

The results from chapter 4 narrow the search space for magics from 2^{64} candidates to numbers in the range $2^{64-w-\mathrm{lsb}(r_{\max})} \leq \alpha < 2^{64-\mathrm{lsb}(r_{\max})}$. Nonetheless there are lots of candidates. We need a fast algorithm for magic testing to make exhaustive search feasible.

5.1. State of the art

Any implementation will have to test the relevant occupancies for non-constructive collisions. It is therefore useful to prepare an array of relevant occupancies and their expected attack sets for fast iteration. In the example code `init_references` uses the Carry-Rippler trick from Section 2.3 to enumerate subsets of the maximum relevant occupancy.

```
typedef struct {
    uint64_t occupied;
    uint64_t attack;
} reference_t;

uint64_t max_relevant_occupancy(int square) {
    int rank = square >> 3, file = square & 7;
    uint64_t edges =
        (((BB_RANKS[0] | BB_RANKS[7]) & ~BB_RANKS[rank]) |
         ((BB_FILES[0] | BB_FILES[7]) & ~BB_FILES[file]));
    return slider_attacks(square, DELTAS, 0) & ~edges;
}

int init_references(reference_t *refs, int square) {
    const uint64_t mask = max_relevant_occupancy(square);
    uint64_t b = 0;
    int size = 0;
    do {
        refs[size].occupied = b;
        refs[size].attack = slider_attacks(square, DELTAS, b);
        size++;
        b = (b - mask) & mask;
    } while (b);
    return size;
}
```

Next a magic candidate can be tested by evaluating the hash function on each relevant occupancy and marking the used buckets in a table. A collision is detected when attempting to put a distinct attack into a non-empty bucket.

```
bool check_magic(uint64_t magic, reference_t *refs, int size) {
    uint64_t table[1 << SHIFT] = { 0 };

    for (int i = 0; i < size; i++) {
        unsigned index = (magic * refs[i].occupied) >> (64 - SHIFT);
        if (!table[index]) {
            table[index] = refs[i].attack;
        } else if (table[index] != refs[i].attack) {
            return false; // collision!
        }
    }

    return true;
}
```

This has to zero-fill a large table for each tested candidate, which is relatively expensive. Stockfish improves upon this by augmenting the table with the age of its entries [8]. Collisions can only be caused by entries of the current generation and the table no longer has to be cleared. Similarly entries could be annotated with the magic candidate they belong to.

```
bool check_magic(uint64_t magic, reference_t *refs, int size) {
    // static, i.e. initialized only once
    static uint64_t table[1 << SHIFT] = { 0 };
    static uint64_t age[1 << SHIFT] = { 0 };

    for (int i = 0; i < size; i++) {
        unsigned index = (magic * refs[i].occupied) >> (64 - SHIFT);
        if (age[index] != magic) {
            // entry belongs to a previous candidate
            // and can be overwritten
            table[index] = refs[i].attack;
            age[index] = magic;
        } else if (table[index] != refs[i].attack) {
            // collision!
            return false;
        }
    }

    return true;
}
```

In principle replacing 64-bit attack sets with 8-bit unique ids or using 32-bit annotations would be more compact, but it is not expected to be faster due to lack of 64-bit alignment.

We also do not have to worry about fine-grained parallelism. Instead we can split the search space into a number of intervals and run separate processes for any one of them, for example using GNU Parallel [27].

5.2. Dynamic ordering

We present a simple but effective improvement.

To refute that a candidate is a magic factor we only have to point out two relevant occupancies that collide non-constructively. In the black box model all occupancies are equally likely to collide, so the best we can do is to cycle through occupancies of distinct attack sets in any order until we find a collision.

Again we find that the multiplicative hash functions do not adhere to the black box model. Some pairs of relevant occupancies are more likely to collide! This is amplified when testing candidates with similar bit patterns, for example linearly scanning a range of candidates.

This dependence on bit patterns can be seen in Figure 5.1. Candidates less than 0x4000200000000 (with many zero bits) seem particularly easy to refute. From then on conventional methods need to test more occupancies on average. As expected the naive zero filling approach is affected similarly, with the constant cost of zero filling added to the generation based version.

We can exploit this by reordering the relevant occupancies in a way that triggers collisions earlier on average. Analytically finding a good order (let alone the best order for a given range of candidates) seems difficult. A simple to implement but effective method is to track collisions for each relevant occupancy dynamically, at runtime.

After testing a number of candidates, references can then be reordered based on the number of collisions. The benchmark in Figure 5.1 reorders occupancies after testing 2^{26} candidates each. More frequent reordering is counter-productive due to the overhead incurred by sorting the 2^{11} relevant occupancies.

This technique seems to be most effective when there are many relevant occupancies. For example Figure 5.2 shows negligible effects for the bishop square **d7** with only 2^5 references. Fortunately testing magics of such squares is relatively faster already.

```c
typedef struct {
    uint64_t occupied;
    uint64_t attack;
    uint64_t collisions; // track number of collisions
} reference_t;
```

```
bool check_magic(uint64_t magic, reference_t *refs, int size) {
    // static, i.e. initialized only once
    static uint64_t table[1 << SHIFT] = { 0 };
    static uint64_t age[1 << SHIFT] = { 0 };

    for (int i = 0; i < size; i++) {
        unsigned index = (magic * refs[i].occupied) >> (64 - SHIFT);
        if (age[index] != magic) {
            table[index] = refs[i].attack;
            age[index] = magic;
        } else if (table[index] != refs[i].attack) {
            refs[i].collisions++; // collision!
            return false;
        }
    }

    return true;
}
```

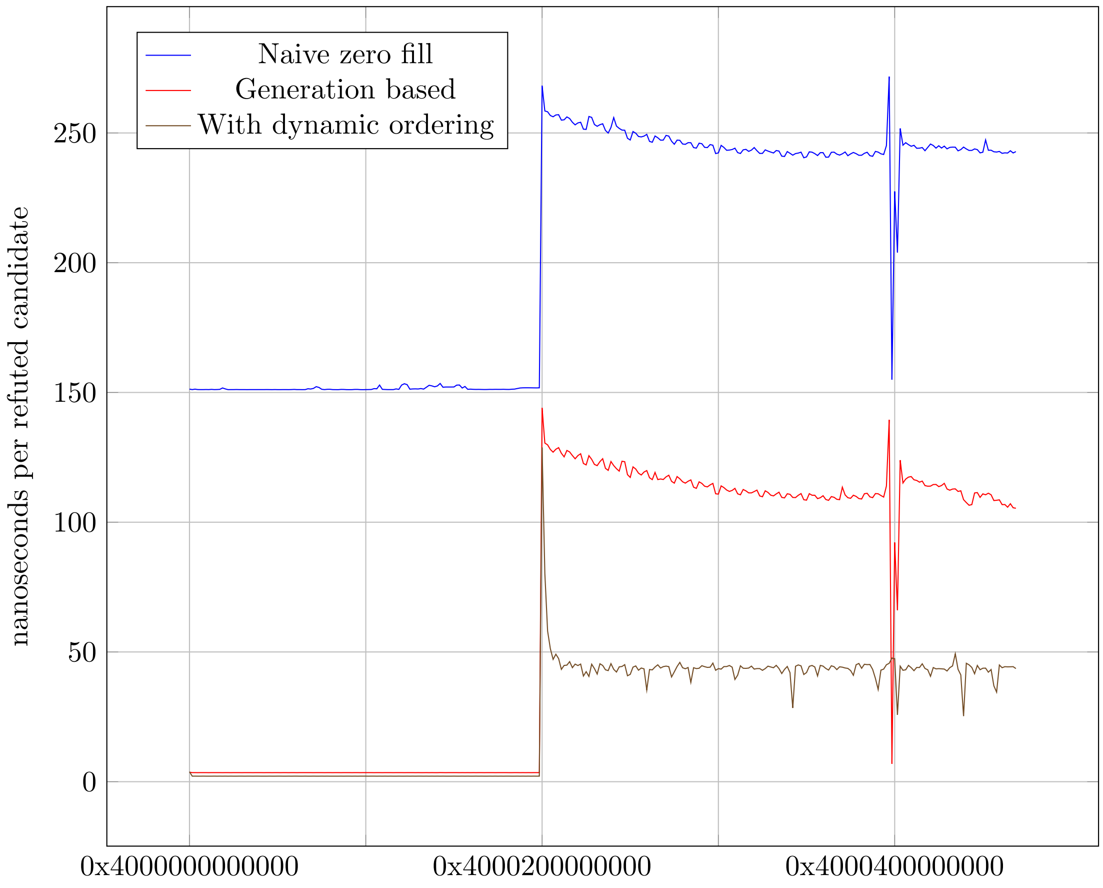


Figure 5.1.: Performance of sequential magic testing for rook square f8 with improved shift 10 in the range for from 0x4000000000000 to 0x40004b0000000

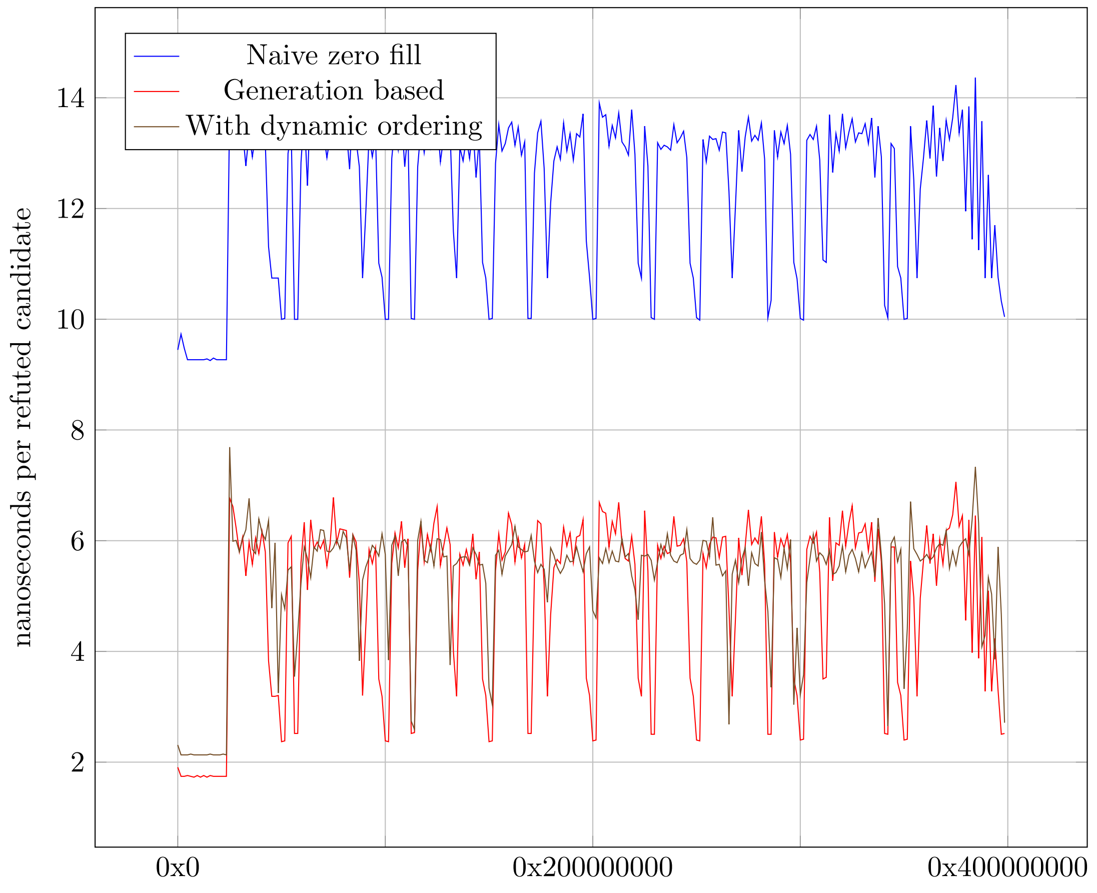


Figure 5.2.: Performance of sequential magic testing for bishop square **d7** with improved shift. Shows the entire range from 0x0 to 0x400000000 with 256 reorderings.

5.3. GPU acceleration

Each magic candidate can be tested independently. We should therefore consider taking advantage of the massive parallelism offered by graphics processors.

Communication between host and graphics card can be minimized by testing groups of magics in chunks. Iterating over a shared table of references is unusual for graphics workloads. This can be alleviated by storing the table as a texture or unrolling the loop into code. We give a listing of the relatively best implementation so far, using CUDA, in Appendix B.

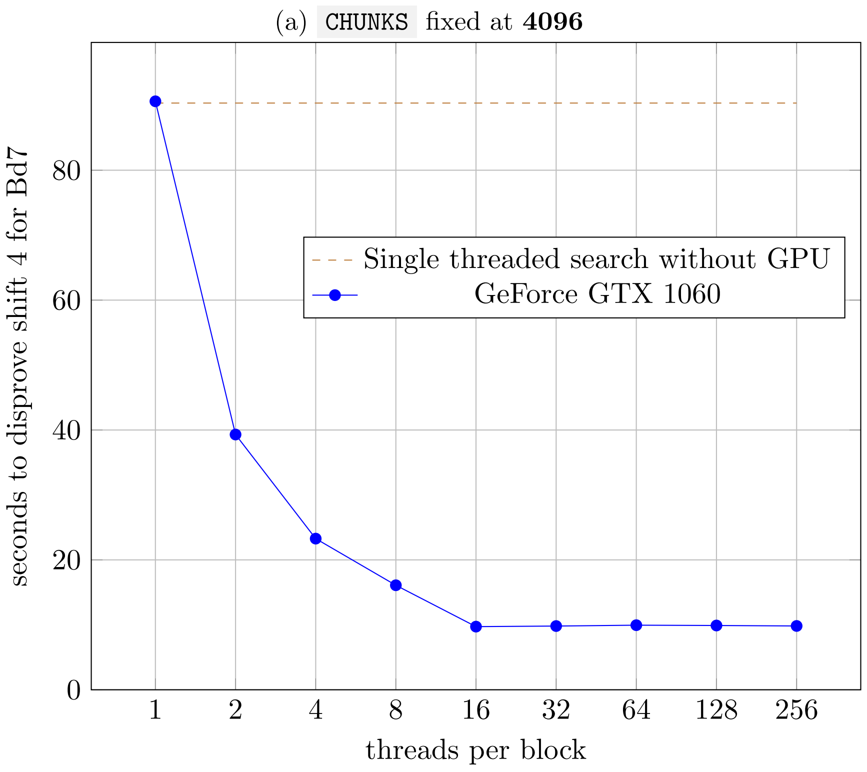
Benchmarks show that the limiting factor is memory bandwidth, i.e. that the advantage over CPUs depends heavily on the size of the scratch space that is required for testing each candidate. On rook squares the table `table` (and possibly `age`) requires at least 2^9 bytes. Here a GeForce GTX 1060 outperforms a single, but not even two, regular CPU cores. On bishop squares with table sizes of only 2^4 bytes a single CPU core is outperformed by a factor of 34.

To achieve these results we use 8 bit unique attack ids instead of 64 bit table entries. Also using the naive zero filling approach performs better than maintaining an additional `age` table.

To select a good number of blocks and threads for the kernel

```
test<<<CHUNKS / THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>
```

we first keep `CHUNKS` fixed at 4096 and vary `THREADS_PER_BLOCK`. Performance does not improve further when choosing more than 32 threads. Next we keep the number of threads per block fixed at 32 and increase the number of chunks. The best result is achieved at the maximum possible value of 2^{19} chunks (16 384 blocks). Details are shown in Figure 5.3.



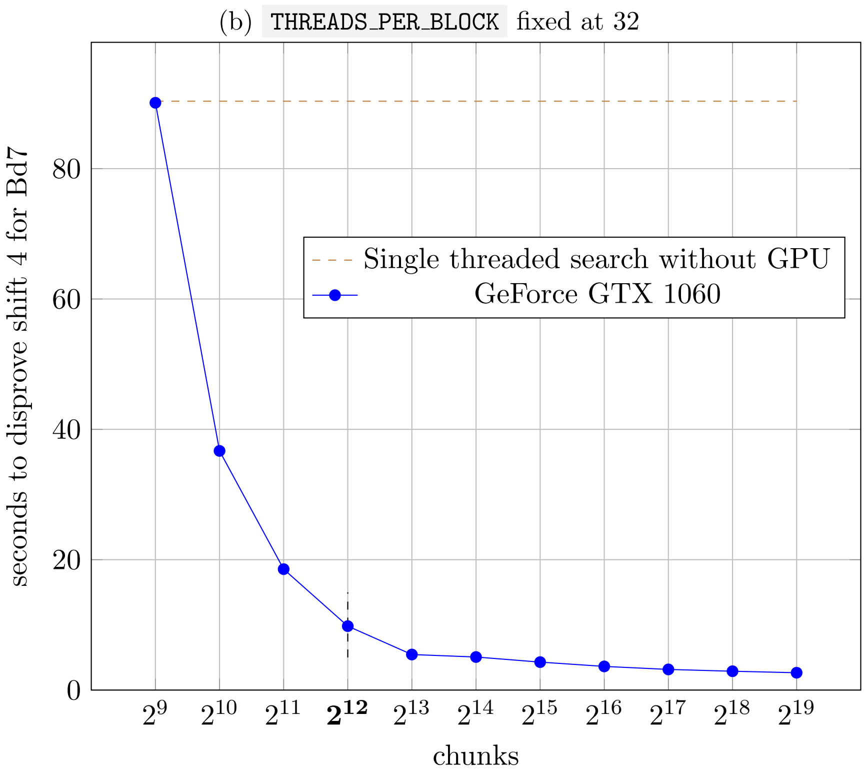


Figure 5.3.: Kernel `test<<<CHUNKS / THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>`, averaged over 5 runs each

6. Results and outlook

In Chapter 3 we discussed state of the art methods for retrieving sliding piece attacks in chess-like games and defined the terms *relevant occupancy* and *magic factor* that are commonly used in chess programming. The benchmarks mentioned in the introduction show that cache efficiency is important for the performance of move generation.

Table 6.1.: State of the art sliding piece generation over 100 runs of Stockfish's benchmark suite measured on an Intel Core i7-6850K CPU @ 3.60GHz

Stockfish	Table size	NPS	±	Speedup
Magic bitboards (x86-64-modern)	861.2kB	2 470 102	10 303	base
Best known fixed shift magics	710.2kB	2 508 235	15 657	1.5%
Best known black magics	703.9kB	2 506 614	10 533	1.5%
PEXT bitboards (x86-64-bmi2)	861.2kB	2 526 798	10 162	2.3%

Previously the best known algorithm for finding good magic factors was to test many random candidates. With a search space of size 2^{64} it was not feasible to definitively disprove the existence of magics with desired properties or find magics with proven minimal index ranges. In Chapter 4 we showed that it is sufficient to search for magic factors α with shift w in the range

$$2^{64-w-\mathrm{lsb}(r_{\max})} \leq \alpha < 2^{64-\mathrm{lsb}(r_{\max})}$$

where $\mathrm{lsb}(r_{\max})$ is the least significant bit of the maximum relevant occupancy. Indeed candidate factors outside of this range periodically describe the same hash function. This significantly reduces the search space to a number of candidates between 2^{26} and 2^{63} depending on piece type and square and is useful when trying to encode additional information in the bits of magic factors.

We also improved the speed of testing individual magic candidates by an order of magnitude for most squares. To do this we reorder relevant occupancies so that tests fail earlier on average, as described in Chapter 5. The full code has been published as a free software project:

`https://github.com/niklasf/magics`

The program is single threaded but can be easily parallelized, for example using a tool like GNU Parallel [27].

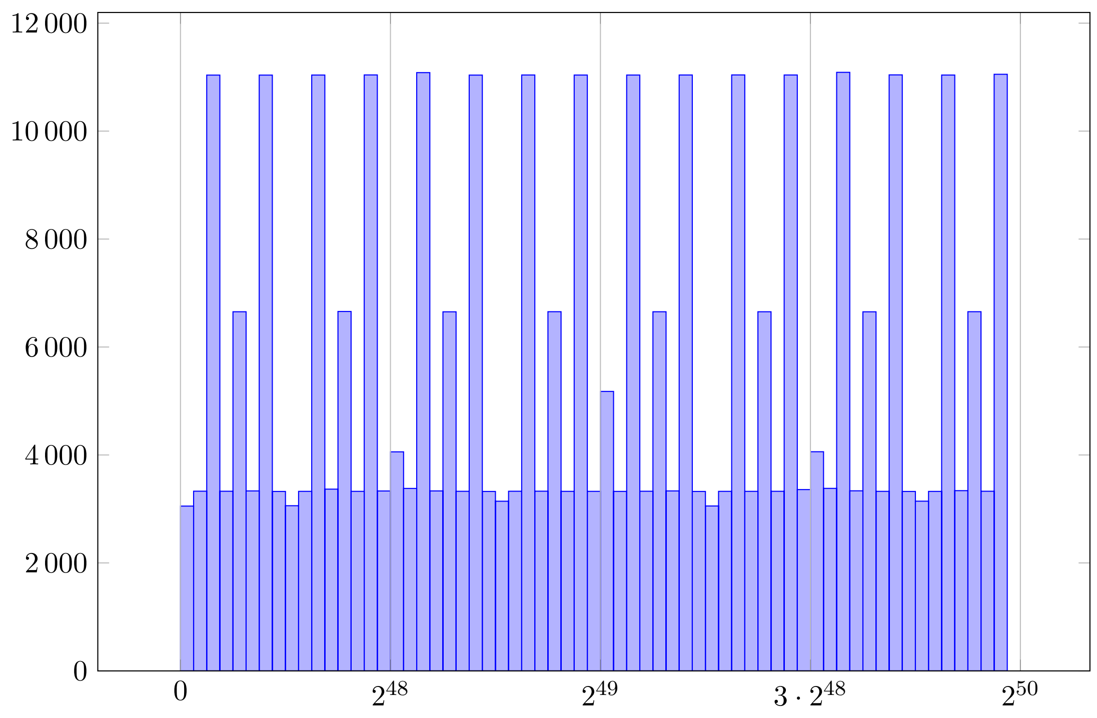


Figure 6.1.: Histogram of bishop magics for d5 (in 64 intervals)

Even now it will take many months to complete the search for optimal magic factors, so it remains to apply the complete results in practice. However there are some notable findings already:

- There is no magic factor with shift 8 for bishops on d5. This means 9 will remain the maximum shift for bishop magics. We can therefore confidently focus on shift 9 when looking for fixed-shift bishop magics.

  Unfortunately there does not seem to be an efficiently checkable certificate for this exhaustive search. Instead we provide a list[1] of 365 475 magic factors with shift 9 (none of which are also shift 8 magics), the program that produced them and claim that this list if exhaustive. The distribution of these magics is shown in Figure 6.1.

- There is no magic factor with shift 10 for rooks on f8. This may be surprising, because previously it seemed easy to find improved rook magics for the seventh and eighth rank (with only a7, g7 and f8 resisting).

- 0x3ff95e5e6a4c0 is a rook magic with shift 10 for g7. This rook square had previously resisted brute-force search.

- Several magics with proven optimal index ranges are given in the appendix.

[1]`https://github.com/niklasf/magics/blob/master/results/bishop-d5.txt?raw=true`

There are some avenues to explore other than finishing the now feasible computations. First of all we cannot exclude that there are further tricks to speed up the search for magic factors. Then with several magic factors for each piece type and square Volker Annuss proposes to build a compact overlapping hash table [4]. Tightly packing hash tables seems useful outside of chess programming. What can be said about this problem and its complexity in general?

Alternatively, it may be interesting to directly model and optimize cache efficiency. This is clearly related to table size, but could take common memory access patterns into account. Such a model may also explain why the currently best known black magics do not outperform the best known fixed shift magics.

A. Best known magics

Thanks to Gerd Isenberg and contributors for collecting the best magics so far [2].

A.1. Rook magics

Rook squares s, their periods $P_{\text{rook}}(s)$ and best known magics with improved shift. New contributions **in bold**. Many more are feasible now, but will still take months to complete.

	s	$P(s)$	c	shift c - 1
h3	23	2^{49}	11	**disproved**
h4	31	2^{49}	11	
h5	39	2^{49}	11	
h6	47	2^{49}	11	**disproved**
h7	55	2^{49}	11	0x510ffff5f63c96a0[a]
h8	63	2^{49}	12	0x7645fffecbfea79e[a]
g3	22	2^{50}	10	
g4	30	2^{50}	10	
g5	38	2^{50}	10	
g6	46	2^{50}	10	
g7	54	2^{50}	10	**0x3ff95e5e6a4c0**
g8	62	2^{50}	10	0x3ffef27eebe74[b]
f3	21	2^{51}	10	
f4	29	2^{51}	10	
f5	37	2^{51}	10	
f6	45	2^{51}	10	
f7	53	2^{51}	10	0xfffffff5fff3e600[c]
f8	61	2^{51}	11	
e3	20	2^{52}	10	
e4	28	2^{52}	10	
e5	36	2^{52}	10	
e6	44	2^{52}	10	
e7	52	2^{52}	10	0xffffffe9ffe7ce00[c]
e8	60	2^{52}	11	0x411fffddffdbf4d6[a]
d3	19	2^{53}	10	
d4	27	2^{53}	10	

	s	$P(s)$	c	shift c - 1
d5	35	2^{53}	10	
d6	43	2^{53}	10	
d7	51	2^{53}	10	0x613fffddffce9200[a]
d8	59	2^{53}	11	0x127fffb9ffdfb5f6[a]
b2	9	2^{54}	10	
c3	18	2^{54}	10	
c4	26	2^{54}	10	
c5	34	2^{54}	10	
c6	42	2^{54}	10	
c7	50	2^{54}	10	0x497fffadff9c2e00[a]
c8	58	2^{54}	11	0x53bfffedffdeb1a2[a]
a2	8	2^{55}	11	
c2	10	2^{55}	10	
d2	11	2^{55}	10	
e2	12	2^{55}	10	
f2	13	2^{55}	10	
g2	14	2^{55}	10	
h2	15	2^{55}	11	
b3	17	2^{55}	10	
b4	25	2^{55}	10	
b5	33	2^{55}	10	
b6	41	2^{55}	10	
b7	49	2^{55}	10	0x48fffe99fecfaa00[a]
b8	57	2^{55}	11	0x61fffeddfeedaeae[a]
a3	16	2^{56}	11	
a4	24	2^{56}	11	
a5	32	2^{56}	11	
a6	40	2^{56}	11	
a7	48	2^{56}	11	0x48fffe99fecfaa00[a]
a8	56	2^{56}	12	0xebffffb9ff9fc526[a]
b1	1	2^{62}	11	
a1	0	2^{63}	12	
c1	2	2^{63}	11	
d1	3	2^{63}	11	
e1	4	2^{63}	11	
f1	5	2^{63}	11	
g1	6	2^{63}	11	
h1	7	2^{63}	12	

[a] Grant Osborne [b] Peter Österlund [c] Volker Annuss

A.2. Bishop magics

Bishop squares s, their periods $P_{\text{bishop}}(\mathbf{s})$ and best known magics or fixed and fancy shifts and their maximum hash values r. New contributions with proven minimal index ranges **in bold**. Other than these easy targets many more are feasible, but computation time spent on rook magics may be more effective to reduce the overall hash table size.

	s	$P(\mathbf{s})$	c	shift 9	r_9	shift c	r_c	shift $c-1$
d8	59	2^{26}	5	**0x84030**	60	**0x208800**	31	**disproved**
e8	60	2^{31}	5	**0x1002020**	31	**0x4f68bcb9**	29	**disproved**
d7	51	2^{34}	5	**0x8403000**	60	**0x20880000**	31	**disproved**
c8	58	2^{34}	5	**0x8208060**	35	**0x50c3417a**	29	**disproved**
e7	52	2^{39}	5	**0x100202000**	31	**0x4f68bcb86d**	29	**disproved**
f8	61	2^{39}	5	**0x40408020**	31	**0x74486419f**	26	**disproved**
h2	15	2^{42}	5	**0x820820020**	61	**0x29748305f5**	22	**0x410509fff0**[a]
d6	43	2^{42}	7	**0x8840200040**	132	**0xa44000800**	127	**disproved**
c7	50	2^{42}	5	**0x820806000**	35	**0x50c34179e6**	29	**disproved**
b8	57	2^{42}	5	**0x820820020**	61	**0x58c328c2ee**	22	**0x1ec04eae595**[a]
g2	14	2^{43}	5	**0x1041040040**	61	**0x52e9060be9**	22	**0x820a13ffe0**[a]
f2	13	2^{44}	5	**0x21c1007bff**	59	**0xa21427af62**	28	**disproved**
e2	12	2^{45}	5	**0x4402000000**	62	**0x8f24760b248**	30	**disproved**
b2	9	2^{46}	5					0xfc087a874a3cf7f6[b]
d2	11	2^{46}	5					**disproved**
a2	8	2^{47}	5					0xfc0846a64a34fff6[b]
c2	10	2^{47}	5					**disproved**
e6	44	2^{47}	7					
f7	53	2^{47}	5					
g8	62	2^{47}	5					0xfc087e8e4bb2f736[b]
h1	7	2^{50}	6					0x7ffdfdfcbd79ffff[b]
h3	23	2^{50}	5					0xfc0a028e5ab4df76[b]
d5	35	2^{50}	9	**0x20080080080**	511	**0x20080080080**	511	**disproved**
c6	42	2^{50}	7					
b7	49	2^{50}	5					0xfc0bf6ce5924f576[b]
a8	56	2^{50}	6					0xfffffcfcfd79edff[b]
g1	6	2^{51}	5					0xfc0a66c64a7ef576[b]
g3	22	2^{51}	5					0x7c0c028f5b34ff76[b]
h4	31	2^{51}	5					
c5	34	2^{51}	7					
b6	41	2^{51}	5					0xf95ffa765afd602b[c]
a7	48	2^{51}	5					0xfc0ff2865334f576[b]
f1	5	2^{52}	5					
f3	21	2^{52}	7					
c4	26	2^{52}	7					
g4	30	2^{52}	5					
b5	33	2^{52}	5					
h5	39	2^{52}	5					
a6	40	2^{52}	5					0xdcefd9b54bfcc09f[c]
e1	4	2^{53}	5					
e3	20	2^{53}	7					
b4	25	2^{53}	5					
f4	29	2^{53}	7					

	s	$P(s)$	c	shift 9	r_9	shift c	r_c	shift $c-1$
a5	32	2^{53}	5					
g5	38	2^{53}	5					
h6	47	2^{53}	5					0x4bffcd8e7c587601[c]
b1	1	2^{54}	5					0xfc0962854a77f576[b]
d1	3	2^{54}	5					
b3	17	2^{54}	5					0x41a01cfad64aaffc[c]
d3	19	2^{54}	7					
a4	24	2^{54}	5					
e4	28	2^{54}	9					
f5	37	2^{54}	7					
g6	46	2^{54}	5					0x43ff9a5cf4ca0c01[b]
h7	55	2^{54}	5					0xc3ff8a54f4ca2c89[b]
a1	0	2^{55}	6					0xffedf9fd7cfcffff[b]
c1	2	2^{55}	5					
a3	16	2^{55}	5					0x73c01af56cf4cffb[c]
c3	18	2^{55}	7					
d4	27	2^{55}	9					
e5	36	2^{55}	9					
f6	45	2^{55}	7					
g7	54	2^{55}	5					0xc3ffb7dc36ca8c89[b]
h8	63	2^{55}	6					0x43ff9e4ef4ca2c89[b]

[a] Another magic first found by Gerd Isenberg is equally good.

[b] Found by Gerd Isenberg.

[c] Found by Richard Pijl.

B. Testing magics with CUDA

This is a listing of the relatively best attempt at GPU accelerated magic testing. Refer to Section 5.3 for a brief discussion.

```
#include <cuda.h>
#include <stdint.h>
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>

#define PERIOD (UINT64_C(1) << UINT64_C(42)) // h2

#define CHUNKS (1 << 19)
#define THREADS_PER_BLOCK 32

__device__ bool check_magic(uint64_t magic) {
    char table[1 << 4] = { 0 };
    int idx;

    idx = (magic * UINT64_C(0)) >> (64 - 4);
    if (table[idx] && table[idx] != 1) return false;
    table[idx] = 1;
    idx = (magic * UINT64_C(524288)) >> (64 - 4);
    if (table[idx] && table[idx] != 2) return false;
    table[idx] = 2;

    // [...] more unrolled tests

    return true;
}

__global__ void test(uint64_t *result) {
    const uint64_t chunk_size = PERIOD / CHUNKS;
    assert(chunk_size * CHUNKS == PERIOD);
    int chunk = THREADS_PER_BLOCK * blockIdx.x + threadIdx.x;

    uint64_t magic = chunk_size * chunk;

    result[chunk] = 0;

    while (magic < chunk_size * (chunk + 1)) {
        if (check_magic(magic)) {
                result[chunk] = magic;
                return;
            }
            magic++;
        }
}

#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr, "GPUassert: %s %s %d\n",
              cudaGetErrorString(code), file, line);
      exit(code);
   }
}

int main() {
    printf("parallel search ...\n");

    uint64_t h_result[CHUNKS] = { 0 };
    uint64_t *d_result;
    gpuErrchk(cudaMalloc(&d_result, sizeof(uint64_t) * CHUNKS));
    test<<<CHUNKS / THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_result);
    gpuErrchk(cudaPeekAtLastError());
    gpuErrchk(cudaDeviceSynchronize());
    gpuErrchk(cudaMemcpy(h_result, d_result,
                         sizeof(uint64_t) * CHUNKS,
                         cudaMemcpyDeviceToHost));

    printf("search complete.\n");

    for (int i = 0; i < CHUNKS; i++) {
        if (h_result[i]) printf("magic: 0x%lx\n", h_result[i]);
    }
}
```

Bibliography

[1] G M Adel'son-Vel'skii et al. "Programming a computer to play chess". In: *Russian Mathematical Surveys* 25.2 (1970), p. 221. URL: http://stacks.iop.org/0036-0279/25/i=2/a=R07.

[2] Gerd Isenberg et al. *Best Magics so far*. 2018. URL: https://chessprogramming.wikispaces.com/Best+Magics+so+far (visited on 04/04/2018).

[3] Volker Annuss. *Black Magic Bitboards*. 2017. URL: http://www.talkchess.com/forum/viewtopic.php?t=64790 (visited on 04/01/2018).

[4] Volker Annuss. *Fixed shift magics with 800KB lookup table*. 2010. URL: http://www.open-aurec.com/wbforum/viewtopic.php?f=4&t=51162 (visited on 03/31/2018).

[5] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. "Hash, Displace, and Compress". In: *Algorithms - ESA 2009*. Ed. by Amos Fiat and Peter Sanders. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 682–693. ISBN: 978-3-642-04128-0.

[6] N.G. de Bruijn. *Acknowledgement of Priority to C. Flye Sainte-Marie on the Counting of Circular Arrangements of 2n Zeros and Ones that Show Each N-letter Word Exactly Once*. TH report // Technische Hogeschool Eindhoven Nederland. Technische Hogeschool, 1975. URL: https://books.google.de/books?id=WrYmOAAACAAJ.

[7] J. Lawrence Carter and Mark N. Wegman. "Universal Classes of Hash Functions (Extended Abstract)". In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: ACM, 1977, pp. 106–112. DOI: 10.1145/800105.803400. URL: http://doi.acm.org/10.1145/800105.803400.

[8] Marco Costalba. *bitboard.cpp*. 2001. URL: https://github.com/official-stockfish/Stockfish/blob/e408fd7b10503/src/bitboard.cpp#L185-L263 (visited on 04/01/2018).

[9] Marco Costalba. *Implement PEXT based attacks*. 2014. URL: https://github.com/official-stockfish/Stockfish/commit/c556fe1d716fcef3 (visited on 04/01/2018).

[10] Martin Dietzfelbinger et al. "A Reliable Randomized Algorithm for the Closest-Pair Problem". In: *J. Algorithms* 25.1 (Oct. 1997), pp. 19–51. ISSN: 0196-6774. DOI: 10.1006/jagm.1997.0873. URL: http://dx.doi.org/10.1006/jagm.1997.0873.

[11] Trevor I. Fenner and Mark Levene. "Move Generation with Perfect Hash Functions". In: *ICGA Journal* 31.1 (2008), pp. 3–12.

[12] Free Software Foundation. *Using the GNU Compiler Collection (GCC): Other Builtins.* 2018. URL: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html (visited on 04/01/2018).

[13] Free Software Foundation. *Using the GNU Compiler Collection (GCC): x86 Built-in Functions.* 2018. URL: https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/x86-Built-in-Functions.html (visited on 04/01/2018).

[14] M. L. Fredman et al. "Storing a sparse table with O(1) worst case access time". In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982).* 1982, pp. 165–169. DOI: 10.1109/SFCS.1982.39.

[15] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading MA, 1988, p. 244. ISBN: 0-201-14236-8.

[16] Lasse Hansen. *Fast(er) bitboard move generator.* 2006. URL: http://www.open-aurec.com/wbforum/viewtopic.php?t=5015 (visited on 03/31/2018).

[17] P. Kannan. *Magic move-bitboard generation in computer chess.* 2007. URL: http://www.pradu.us/old/Nov27_2008/Buzz/research/magic/Bitboards.pdf (visited on 03/20/2018).

[18] Marcel van Kervinck. *Re: move generators in computer chess.* 1994. URL: https://groups.google.com/forum/#!msg/rec.games.chess/KnJvBnhgDKU/yCi5yBx18PQJ (visited on 04/01/2018).

[19] Eugene Kotlov. *Magic number comprising offset.* 2018. URL: http://www.talkchess.com/forum/viewtopic.php?t=66538 (visited on 04/01/2018).

[20] Maurizio Monge. "On perfect hashing of numbers with sparse digit representation via multiplication by a constant". In: *Discrete Applied Mathematics* 159.11 (2011), pp. 1176–1179. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2011.03.007. URL: http://www.sciencedirect.com/science/article/pii/S0166218X11000837.

[21] Eugene Nalimov. *Re: Will the Itanium have a BSF or BSR instruction?* 2000. URL: https://www.stmintz.com/ccc/index.php?id=124712 (visited on 04/01/2018).

[22] Long Hoang Nguyen and A. W. Roscoe. "Short-Output Universal Hash Functions and Their Use in Fast and Secure Data Authentication". In: *Fast Software Encryption.* Ed. by Anne Canteaut. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 326–345. ISBN: 978-3-642-34047-5.

[23] Grant Osborne. *Incorporating the shift into the magic number.* 2008. URL: http://www.talkchess.com/forum/viewtopic.php?topic_view=threads&p=196157&t=21329 (visited on 04/01/2018).

[24] Rasmus Pagh. "Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions". In: *Algorithms and Data Structures.* Ed. by Frank Dehne et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 49–54. ISBN: 978-3-540-48447-9.

[25] Fritz Reul. "New Architectures in Computer Chess". In: *TICC Dissertation Series 6* (2009), pp. 53–68. URL: https://pure.uvt.nl/ws/files/1098572/Proefschrift_Fritz_Reul_170609.pdf.

[26] Michael Sherwin. *Magic Bitboards Explained!* 2006. URL: http://www.open-aurec.com/wbforum/viewtopic.php?f=4&t=5958 (visited on 03/31/2018).

[27] Ole Tange. *GNU Parallel 2018*. Ole Tange, Apr. 2018. DOI: 10.5281/zenodo.1146014. URL: https://doi.org/10.5281/zenodo.1146014.

[28] Kim Walisch. *Bitscan forward: De Bruijn Multiplication with separated LS1B*. 2012. URL: https://chessprogramming.wikispaces.com/BitScan#Bitscan%20forward-De%20Bruijn%20Multiplication-With%20separated%20LS1B (visited on 04/01/2018).

[29] Zach Wegner. *Haswell New Instructions*. 2011. URL: http://www.talkchess.com/forum/viewtopic.php?t=40333 (visited on 03/31/2018).