Preprint of an article in: ICCA Journal 20(3), pp. 166–176, Sept. 1997.

How DarkThought Plays Chess

Ernst A. Heinz*
Institute for Program Structures and Data Organization (IPD)
School of Computer Science, University of Karlsruhe
P.O. Box 6980, D-76128 Karlsruhe, F.R. Germany

September 10, 1997

Abstract

DarkThought[1] is a bitboard-based chess program developed at the University of Karlsruhe that has successfully participated in all world championships since 1995. On a 500MHz DEC Alpha-21164a with 128MB RAM, DarkThought routinely reaches speeds of 200K nps (nodes per second) in the middlegame while peaking at over 650K nps in the endgame.

The article describes the design and inner structure of DarkThought. To this end, it presents detailed accounts of the chess engine while elaborating on some of its innovations: *rotated bitboards*, a *fully programmable* leaf-node evaluation function, and versatile *search parameterization*.

1 Introduction

Since entering competition in May 1995, the chess program DarkThought has participated in three ICCA world championships and two AEGON man vs. machine tournaments. Work on the program started about two years before its first tournament game, when Peter W. Gillgasch and Ernst A. Heinz founded IPD's computer chess group in mid-1993. Markus Gille joined the group in early 1994 and DarkThought proceeded as a joint development of all three until Peter left the team in April 1996.

The central design philosophy of DarkThought stresses a balanced combination of speed and knowledge that some other strong chess programs also seem to pursue. DarkThought is neither heavily knowledge-based nor dumb and fast only – instead, it complements a speedy search with a fairly knowledgable evaluation. The successful tournament history of DarkThought (see Appendix B) and its world-class test suite results for BT-2630 with a rating of 2494 ELO and LCT-II with a rating of 2535 ELO (see Appendix A) strongly support the validity of the combined approach.

Other important design principles of DarkThought are configurability and portability. The program is written in ANSI-C and runs on various Unix platforms (e.g. DEC Alpha, HP PA-RISC, Intel x86, and Sun Sparc). Special

*Email = <heinze@ira.uka.de>, WWW = <http://wwwipd.ira.uka.de/~heinze/>
[1]WWW = <http://wwwipd.ira.uka.de/Tichy/DarkThought/>

source-code switches allow for the seamless selection of different implementations of time-critical functions. Thus, DarkThought can easily be tailored for any desired platform (e.g. 32-bit versus 64-bit CPUs). Other switches create different search "personalities" without any loss of efficiency, thereby enabling frequent comparisons between the most interesting ones.

Today, DarkThought is completely bitboard-based although it did not start as such. The elegance and performance potential of bitboards has lured researchers since the early days of computer chess. Several historically famous chess programs used bitboards while running on mainframes or vector supercomputers (e.g. Kaissa [1], Cray Blitz [20], Chess 4.5 [35]). The advent of ever faster microprocessors with ever larger natural word lengths made the bitboard approach more and more attractive for microcomputer-based chess programs during the past years. Beside Crafty and DarkThought there are now quite a few bitboard chess-programs around that have not yet emerged in public competition but will surely do so in the future. Furthermore, general interest in bitboards has increased significantly since Intel announced its next-generation 64-bit microprocessor to debut in 1998.

The remainder of this article firstly portrays a brief implementation history of DarkThought in Section 2. Then it details the most important parts of the program: the bitboard engine in Section 3, the search engine in Section 4, and the evaluation engine in Section 5. Last but not least, the article concludes with an outlook on future work.

2 Implementation History

Up to now, there exist four major DarkThought versions. The first one is a direct offspring of a chess program in Pascal that Peter had written for his 25MHz 68000 Macintosh in 1992. After porting the program to ANSI-C on IPD's DEC Alpha workstations, incrementally updated piece bitboards were added to speed up the evolving evaluation routines. This hybrid version of DarkThought combined a traditional move generator and search engine that manipulated piece lists with a bitboard-based evaluation. It participated in the 8th WCCC (Hong Kong, May 1995), the 13th WMCC (Paderborn, October 1995), and the 11th AEGON (Den Haag, April 1996) while running at 20K – 40K nps on a 275MHz DEC Alpha-21064a.

The second DarkThought version was already completely bitboard-based and as such a full-blown 64-bit program. But it never played a full game of chess because it could only handle test suites and lacked many important features that are necessary for real play. Peter started to work on the new program in June 1995 immediately after returning from the 8th WCCC in Hong Kong while Ernst and Markus still tuned the old DarkThought for the upcoming 13th WMCC. Beside an extensive bitboard infrastructure, the second DarkThought also featured *rotated bitboards* for sliding pieces. Peter implemented them independently of Robert Hyatt's Crafty efforts and – to the best of our knowledge – DarkThought was the first chess program ever to contain a working implementation thereof.

The third DarkThought version emerged after Peter left the team in April 1996. While the bitboard move-generator of the second version remained mostly unchanged, Ernst and Markus rewrote large parts of the bitboard infrastructure, redesigned the search engine completely, and added an innovative evaluation engine with a *fully programmable* leaf-node evaluation function. Successive refinements of the third DarkThought version participated in the 14th WMCC (Jakarta, October 1996) and the 12th AEGON (Den Haag, April 1997).

The fourth and now current DarkThought version is an evolution of its predecessor featuring lots of speed optimizations as well as dramatic changes in the bitboard move-generator and various fundamental data structures (e.g. new chess-board representation). On a 500MHz DEC Alpha-21164a, the DarkThought version as of July 31, 1997 routinely reaches 200K nps in the middlegame while peaking at over 650K nps in the endgame (see Appendix A).

3 Bitboard Engine

The advantages of bitboards in computer chess spring from the fact that trivial bitwise operations on them suffice to answer interesting questions about pieces on a chess board. For example, imagine a bitboard with 1-bits representing all squares attacked by white Pawns and another bitboard with 1-bits indicating all black pieces. A bitwise AND operation combining both these bitboards yields a result bitboard with 1-bits for all black pieces attacked by white Pawns. Because computers normally excel at bitwise operations while performing them on all bits of their natural word length simultaneously, bitboard usage proves to be very efficient.

The early proponents of the bitboard approach introduced all the basic principles of bitboard-based move generation, piece-attack detection, and positional evaluation [1, 14, 20, 35, 37]. Their publications contain a wealth of information about traditional bitboard techniques that will not be repeated here. Instead, this article focusses on the bitboard infrastructure and the fundamental principles of rotated bitboards as pioneered by DarkThought.

3.1 Bitboard Infrastructure

DarkThought makes extensive use of bitboards stored as 64-bit unsigned integers where each bit represents a square of the chess board numbered from a8 = 63 to h1 = 0. Because the bit length of integers differs between compilers, machines, and operating systems, DarkThought defines a set of portable data types like `unsigned_8`, ..., `unsigned_64` that are automatically mapped to the appropriate types of the target platform (e.g. `unsigned_64` defaults to `unsigned long long` in GNU C).

In order to simplify the handling of bitboards, DarkThought contains an abundant collection of functions and macros that provide efficient implementations of all the needed bitboard operations. Beside the basic bitwise AND, OR, XOR etc. the collection includes cascaded combinations thereof because they are frequently used and may even be executed in one cycle on some machines. The expression `(x & ~y)`,[2] for instance, corresponds to a single DEC Alpha instruction. Other important bitboard operations enjoy marvelously simple yet non-obvious formulations due to the wonders of binary subtraction and the two's complement. Two famous members of this class are the expressions `(b & -b)` and `(b & (b - 1))`. The first clears all but the least significant 1-bit of a bitboard while the second clears only the least significant 1-bit.

Unfortunately, the central find-bit and population-count operations have higher algorithmic complexity. Although a few CPUs like Motorola PowerPC, Sun UltraSparc and the forthcoming DEC Alpha-21264 feature instruction-level support of these operations, most current machines still require software implementations. The efficiency of the implementations in terms of speed and memory consumption is of crucial importance for bitboard-based chess programs because it tends to limit their overall performance. In this respect, the short expression for clearing the least significant 1-bit gives rise to a neat formulation of an iterative population-count function.

```
unsigned int iterative_popcount(unsigned_64 b) {
    unsigned int n;
    for (n = 0; b != 0; n++, b &= (b - 1));
    return n;
}
```

If the given bitboard contains few 1-bits, the loop terminates quickly thus making this short and easily inlinable function run fast. Otherwise, DarkThought prefers the following non-iterative formulation that stems from the well-known "Hacker's Memory" collection of programming tricks. It performs better than intuitive methods with lookup tables because the tables get either too large or need too many lookups.[3]

```
#define m1 ((unsigned_64) 0x5555555555555555)
#define m2 ((unsigned_64) 0x3333333333333333)

unsigned int non_iterative_popcount(const unsigned_64 b) {
    unsigned_32 n;
    const unsigned_64 a = b - ((b >> 1) & m1);
    const unsigned_64 c = (a & m2) + ((a >> 2) & m2);
    n = ((unsigned_32) c) + ((unsigned_32) (c >> 32));
    n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F);
    n = (n & 0xFFFF) + (n >> 16);
    n = (n & 0xFF) + (n >> 8);
    return n;
}
```

The idea of this population-count trick can also be applied to the find-bit problem yielding a find-function without any memory accesses. Unfortunately, however, this find-bit function is slower than fine-tuned table lookups. As executed

[2] ANSI-C syntax is used for all examples throughout the article.

[3] A clear explanation of how this bit-fiddling trick works lies beyond the scope of the article.

by DarkThought the lookups generate negligible memory traffic and compile to roughly 15 machine instructions on a DEC Alpha while relying on the conditional data-move capabilities of modern CPUs. This clever find-bit implementation further exploits the fact that the number of the least significant 1-bit of a bitboard is equal to the number of the most significant 1-bit of the bitboard xor'ed with itself minus one: `FIND_LSB(b)` = `FIND_MSB(b ^ (b - 1))`.

3.2 Rotated Bitboards

Vectors of attack masks for all pieces on each square of an otherwise empty chess board enable an easy attack detection for sliding pieces without having to loop over the squares of the respective diagonals, files, and ranks [37]. This loopless attack-detection scheme operates on consecutive strings of bits representing the rays of squares potentially covered by the sliding piece. In order to really excel, it therefore requires efficient access to consecutive strings of ray bits. As listed below, the normal row-major mapping of squares to bits in a bitboard consecutively aligns the bits of each file. Hence, it only provides efficient access to consecutive strings of file bits.

Normal Bitboard.

#7	#6	#5	#4	#3	#2	#1	#0	Bit/Byte
a8	b8	c8	d8	e8	f8	g8	h8	**#7**
a7	b7	c7	d7	e7	f7	g7	h7	**#6**
a6	b6	c6	d6	e6	f6	g6	h6	**#5**
a5	b5	c5	d5	e5	f5	g5	h5	**#4**
a4	b4	c4	d4	e4	f4	g4	h4	**#3**
a3	b3	c3	d3	e3	f3	g3	h3	**#2**
a2	b2	c2	d2	e2	f2	g2	h2	**#1**
a1	b1	c1	d1	e1	f1	g1	h1	**#0**

A column-major mapping of squares to bits in a bitboard, however, consecutively aligns the bits of each rank. Consequently, such flipped bitboards allow for efficient access to consecutive strings of rank bits.

Flipped Bitboard.

#7	#6	#5	#4	#3	#2	#1	#0	Bit/Byte
a8	a7	a6	a5	a4	a3	a2	a1	**#7**
b8	b7	b6	b5	b4	b3	b2	b1	**#6**
c8	c7	c6	c5	c4	c3	c2	c1	**#5**
d8	d7	d6	d5	d4	d3	d2	d1	**#4**
e8	e7	e6	e5	e4	e3	e2	e1	**#3**
f8	f7	f6	f5	f4	f3	f2	f1	**#2**
g8	g7	g6	g5	g4	g3	g2	g1	**#1**
h8	h7	h6	h5	h4	h3	h2	h1	**#0**

In the case of diagonals, the necessary mappings prove to be more complex. The number of diagonals amounts to 15 in each of the two directions a1–h8 and a8–h1. Furthermore, diagonals are of variable lengths ranging from 1 to 8. Because not all diagonals can be stuffed into separate bytes of a bitboard the mapping must pack some together in a clever way.

Figuratively, viable mappings can be deduced as follows: slice a normal bitboard along one of the main diagonals (a1–h8 or a8–h1), then rotate the half with the squares of the main diagonal to lie at the bottom, and finally paste the other half such that it results in a parallelogram. As shown below, the diagonalized mappings of squares to bits in a bitboard consecutively align the bits of each a1–h8 and a8–h1 diagonal. Thus, diagonalized bitboards enable efficient access to consecutive strings of diagonal bits if the ends of the diagonals (as marked by vertical bars) are known.

A1-H8 Bitboard.

#7	#6	#5	#4	#3	#2	#1	#0	Bit/Byte
a8	∣ b1	c2	d3	e4	f5	g6	h7	**#7**
a7	b8	∣ c1	d2	e3	f4	g5	h6	**#6**
a6	b7	c8	∣ d1	e2	f3	g4	h5	**#5**
a5	b6	c7	d8	∣ e1	f2	g3	h4	**#4**
a4	b5	c6	d7	e8	∣ f1	g2	h3	**#3**
a3	b4	c5	d6	e7	f8	∣ g1	h2	**#2**
a2	b3	c4	d5	e6	f7	g8	∣ h1	**#1**
a1	b2	c3	d4	e5	f6	g7	h8	**#0**

A8-H1 Bitboard.

#7	#6	#5	#4	#3	#2	#1	#0	Bit/Byte
a8	b7	c6	d5	e4	f3	g2	h1	**#7**
a7	b6	c5	d4	e3	f2	g1	∣ h8	**#6**
a6	b5	c4	d3	e2	f1	∣ g8	h7	**#5**
a5	b4	c3	d2	e1	∣ f8	g7	h6	**#4**
a4	b3	c2	d1	∣ e8	f7	g6	h5	**#3**
a3	b2	c1	∣ d8	e7	f6	g5	h4	**#2**
a2	b1	∣ c8	d7	e6	f5	g4	h3	**#1**
a1	∣ b8	c7	d6	e5	f4	g3	h2	**#0**

For historical reasons, diagonalized and flipped bitboards are both called rotated. In order to fully exploit the benefits of rotation, DarkThought incrementally updates rotated bitboards with 1-bits for all occupied squares of the current position. Additionally, it manages compressed vectors of rotated attack masks that strike a good balance between instruction and memory efficiency.

4 Search Engine

DarkThought is a sophisticated alpha-beta searcher using PVS/NegaScout [13, 28] with state-of-the-art enhancements like aggressive futility pruning [19, 31], internal iterative deepening [3, 33], dynamic move ordering (history+killer heuristic) [2, 16, 30, 32, 35], recursive null-move pruning [7, 15, 17], selective extensions [3, 6], interior-node recognizers [18, 34], and an extended transposition table [26, 35]. On average, all enhancements taken together reduce the effective branching factor of DarkThought to 2–3 and its search-tree size to roughly 55% of that of the according minimal tree [21]. These observations are consistent with reports by other researchers [6, 38].

At the top level DarkThought employs lazy alpha bounding and iterative deepening with an aspiration window of half a Pawn [22, 35]. In contrast to plain alpha bounding [31], the lazy scheme delays the complete resolution of both new best moves and fail highs up to the next iteration. Top-level alpha bounding often saves some effort while at the same time searching new best moves one ply deeper than usual. If the top-level search is unstable or if the final score lies far below the score of the previous move, DarkThought extends the duration of the current search up to threefold when playing in tournament mode.

The full-width search of variable depth is backed by a "most valuable victim / least valuable aggressor (MVV/LVA)" style quiescence [11] that triggers the static leaf-node evaluations and follows capturing moves only. In order to work well such half-blind quiescence needs deep full-width searches as executed by DarkThought which routinely reaches 11–13 plies in standard middlegame positions within a minute. Due to its high search depths and wealth of extensions, DarkThought does not switch off null moves until one side has no more pieces (i.e., it still performs null-move pruning if both sides have just a single minor piece plus Pawns). Our overall experience of numerous matches and test-suite runs suggests that the positive effects of largely increased full-width depth strongly outweigh the zugzwang-related negative ones. At least for tournament play, early discovery of deep pawn breakthroughs and related tactical tricks seems to be much more important than accurate zugzwang detection in endgames with major or minor pieces on both sides.

4.1 Node Expansion

DarkThought implements a standard node-expansion procedure whose structure probably does not differ much from that of other chess programs. It executes the following successive steps as suggested by several authors and excellently summarized in [24, 25].

- Extension decisions related to incoming move (U),
- internal iterative deepening (U),
- transposition-table lookup (L/U),
- interior-node recognizer decision (L/U),
- endgame database lookup (L/U),
- null-move search with deep extension decision (U),
- single-reply extension decision (U),
- futility pruning decision (F/L),
- static evaluation (F/L),
- search of all outgoing moves (L/U).

The letters in parantheses behind each step explain where in the search tree it is actually applied: F means "frontier" (depth = 1), L "lower part" (depth ≤ 0), and U "upper part" (depth ≥ 1) where depth denotes ply distance from the quiescence horizon. The last node expansion step achieves good move ordering by arranging the sequence of recursive search calls as listed below.

1. Hashed move from transposition table (L/U),
2. capture moves in MVV/LVA order (L/U),
3. killer moves (U),
4. history moves (U),
5. statically pre-sorted remaining moves (U).

This ordering scheme enjoys wide-spread use in computer chess because it represents the best currently known according to many independent researchers. An unintended yet nice side effect thereof is that it also saves precious memory bandwidth because it does not require the creation of a move list prior to history-move selection.

In contrast to most other chess programs, DarkThought uses a single node-expansion routine to handle both full-width and quiescence-search nodes. Because of better code reuse (see above lists of steps), a single node expansion routine can be made smaller than two separate ones and it is much easier to reconfigure. The complexity of the implementation, however, increases considerably as compared to the solution with two separate routines.

4.2 Extension Heuristics

At the time of this writing, DarkThought relies on the selective deepening powers of five different search-extension heuristics.

- Check extensions [35],
- deep-search extensions [15],
- recapture extensions [10],
- Pawn extensions [23],
- single-reply extensions [23].

In order to avoid possibly explosive growth of the search tree as caused by excessive deep-search extensions in the case of repeated mutual mating threats, DarkThought restricts them to null moves at depth = 2 in the first $2 * \textit{iteration\_number}$ plies on all paths. For recapture extensions DarkThought employs a Hitech-style definition based on material balance instead of full evaluation scores: recaptures are captures or Pawn promotions that lead to a similar material balance as that of the root node. Pawn extensions relate to moves of advanced Pawns to the 7th / 2nd rank (for White / Black resp.) and all non-capturing moves of Pawns to the 6th / 3rd rank.

According to literally thousands of test runs these extensions combine extremely well with one another and the rest of the search, especially with futility and null-move pruning (see [6] for an independent confirmation). Still, DarkThought attempts to reduce the number of meaningless extension triggers by restricting some of them (e.g. Pawn extensions) to the lower parts of the search tree up to a certain, possibly adaptive depth threshold. Such *extension layering* requires careful fine-tuning before it works without intolerable reduction of the overall extension benefits. Fortunately, this seems to be its only disadvantage.

4.3 Search Parameterization

As mentioned in the introduction, one important design principle of DarkThought is configurability without loss of efficiency. To this end, it contains a large collection of source-code switches that allow for the quick and easy generation of versatile search "personalities". Descriptions of the most interesting parameters and their effects follow below.

Search Algorithm. Beside standard PVS/NegaScout, DarkThought can run pure fail-soft alpha-beta (FAB) and a mixture of both that uses PVS until reaching a configurable depth where it switches to FAB. Viable values for the depth threshold range from 1 to 4.

Null-Move Search. Generally regarded as both a curse and a blessing of microcomputer-based chess programs, the realm of null-move pruning provides ample opportunities for many improvements (e.g. by parameter fine-tuning). In DarkThought, null moves may be layered (i.e. disabled in either the upper or lower parts of the search tree up to a predefined depth threshold), made non-recursive or even switched off completely. The first DarkThought delayed null moves until depths ≤ 5 after Don Dailey (of Socrates and StarSocrates fame) enthused about this scheme during the 8th WCCC in Hong Kong, 1995.

Like many other chess programs, DarkThought currently uses a depth-reduction factor of 2 for null-move searches. But other values (especially depth-adaptive ones in combination with various material thresholds for disabling the null move in endgames) are constantly experimented with. The same holds for depth and score margin of the deep-search extension which default to 2 and the value of 3 Pawns respectively.

Transposition-Table Access. DarkThought performs multiple probes in the transposition table because experiments conducted by Peter in 1993 showed the superiority thereof as compared to multiple tables (see [5] for an independent confirmation). The number of probes per access is a redefinable parameter normally set to 4.

The depth up to which DarkThought accesses the transposition table can be configured independently for read and write operations. Setting both to one ply below the quiescence horizon currently produces the best overall results.

Extension Heuristics. All extensions may individually be layered or completely turned off. Check extensions can be disabled if the piece material of the checking side falls below a certain threshold (e.g. less than two pieces). The depth up to which checking moves are followed unconditionally is modifiable, too, yet normally set to the frontier as this combines well with futility pruning and capture-only quiescence.

Data Structures. Because most computers exhibit different behaviour as for memory performance and cache sensitivity, DarkThought contains distinct implementations of many data structures accessed during the search. First and foremost, you may choose between a single global board modified by corresponding make/unmake-move operations and stacked local boards that require no unmake after but copying prior to the make-move operation during node expansion. Similarly, stacked local move lists as opposed to a single global move list may be selected. Although they feature much higher memory consumption, the localized data structures will simplify any future parallelization attempts. Last but not least, there exist several variations of history tables with the piece-to-square formulation [36] used by default.

5 Evaluation Engine

DarkThought decouples the chess knowledge as specialized scoring functions from the rest of the evaluation infrastructure. Before the start of each search, a root-node oracle [10] not only fills all piece-on-square tables (PSTs) afresh but it also selects appropriate scoring functions according to an extensive analysis of the root position. For each selected scoring function the oracle compiles a list of associated parameters and plugs both into the fully root-node programmable leaf-node evaluation function. In contrast thereto, limited hardware capabilities severely restricted the programmability of the evaluation in Hitech [9] that inspired the design of the evaluation engine in DarkThought.

The evaluation infrastructure automatically tracks the state of the evaluation function (incl. PSTs) in order to detect any changes incurred by the oracle. In case of a change, the evaluation-dependent contents of all affected hash tables are cleared. Other important services of the evaluation infrastructure include the root-node programming interface and all-out management of the evaluation hash tables. Thus, the scoring functions can concentrate on their respective tasks without ever worrying or even knowing about hashing.

5.1 Programmable Evaluation Function

Unlike other chess programs, DarkThought features a leaf-node evaluation function whose properties are almost completely root-node programmable. Beside oracle-supplied PST values, this includes upper score bounds for futility-pruning decisions and plug-in functions for all remaining scoring tasks. The interplay of the programmable components is hard-wired inside the evaluation infrastructure and proceeds as follows.

- Material-balance scoring,
- evaluation futility-pruning decision,
- Pawn-structure scoring (H),
- Pawn-related King scoring (H),
- race scoring of passed Pawns,
- PST piece-placement scoring,
- positional futility-pruning decision,
- further positional scoring (H).

For all steps marked with an H, the evaluation infrastructure automatically hashes the computed scores. While hashing of Pawn-structure scores and Pawn-related King scores works most efficiently with separate hash tables [10, 26], the positional as well as the overall evaluation scores get stored in the extended transposition table because they relate to the entire board position. Hashing of material-balance scores was abandoned after none of several examined hash functions achieved acceptable hit rates.

After the completion of Pawn-structure scoring all important facts related to the bare Pawn constellation of the current position are known. The evaluation infrastructure automatically hashes these facts, too, before passing them to the subsequent scoring steps for efficient reuse.

The futility-pruning decisions provide quick exits if the sum of partial scores so far plus an upper bound for the remaining scores falls below alpha. The oracle determines appropriate upper bounds (one for the whole score excluding material balance and one for the positional score alone) with respect to the material situation at the root position and the programmed scoring plug-ins.

The root-node programming process itself includes the selection and installation of appropriate evaluation functions with associated parameter tables for material-balance scoring, Pawn-structure scoring, Pawn-related King scoring, Pawn-race scoring, and further positional scoring. Up to now, the oracle can choose between general functions for Pawn-structure analysis, Pawn-race analysis, and complex positional analysis as well as a special function for middlegame King-safety analysis. These functions know about fianchetto Bishops, center control, connected Rooks, King attacks, mobility, open files, outposts, passed Pawns, Pawn shelters, Pawn storms, and weak Pawns. More special scoring functions for different endgames and general functions for the analysis of positions with reduced material will be implemented in the future.

5.2 Evaluation Machines

Taken together, all programmable components of the evaluation function constitute an *evaluation machine* that describes the current state of the whole evaluation engine. Evaluation machines are the central data structures which the oracle and the evaluation infrastructure operate on. DarkThought specifies them by the following EVAL_MACHINE type declaration.

```
typedef struct {

    int (*material_balance)(const SIDE, const BOARD *const);

    int (*analyze_pawns)(const BOARD *const, PAWN_FACTS *const);

    int (*analyze_king)(const SIDE, const BOARD *const,
                                    const PAWN_FACTS *const);
    int (*analyze_race)(const SIDE, const BOARD *const,
                                    const PAWN_FACTS *const);
    int (*analyze_posn)(const SIDE, const BOARD *const,
                                    const PAWN_FACTS *const);
    int max_eval_score[2], max_posn_score[2];

    void *king_tables[2], *pawn_tables[2], *posn_tables[2];
    int king_tab_size[2], pawn_tab_size[2], posn_tab_size[2];

    int piece_on_square_table[6][64][2];

} EVAL_MACHINE;
```

The five function variables material_balance, analyze_pawns, analyze_king, analyze_race, and analyze_posn refer to the currently installed scoring plug-ins. The type identifiers BOARD, SIDE, and PAWN_FACTS denote the internal representations of the chess board, of the side to move, and of the facts related to the bare Pawn constellation respectively. Please note that analyze_pawns must compute the Pawn facts because all remaining analysis functions receive them as constant read-only parameters.

The vectors max_eval_score and max_posn_score contain the upper bound values as determined by the oracle and needed by the evaluation as well as the positional futility-pruning decisions for both sides. Similarly, the vectors king_tables, pawn_tables, and posn_tables hold pointers to the parameter tables of the corresponding scoring functions for both sides. The actual sizes of the parameter tables are given by the respective element values of the vectors king_tab_size, pawn_tab_size, and posn_tab_size. Last but not least, the vector piece_on_square_table contains the placement scores of all piece types on any square of the chess board for both sides as calculated by the oracle.

6 Future Work

One of the nicest things about computer chess is the fact that the to-do list of interesting ideas seems to grow steadily despite all the improvements made since the last time you checked. For DarkThought, in particular, the whole field of parallelization is still untapped. While parallel or distributed probability-based B* [8] looks very attractive, DarkThought already contains all localized data structures as necessary for massive alpha-beta parallelization [12].

DarkThought on a single processor will use fractional ply extensions [23] in the foreseeable future. Singular extensions [4], enhanced transposition-table cutoffs [29], and MTD(f) [27] are all worth a trial, too. Automatic learning and tuning of opening books ranks high on the list as well.

More mundane tasks like low-level speed tuning and similar programming "hacks" are scheduled to take place just before important tournaments such as world championships. Please visit the WWW page of DarkThought at URL <http://wwwipd.ira.uka.de/Tichy/DarkThought/> in order to follow its progress and down-load further information (e.g. notations of all tournament games played by the program).

7 Acknowledgements

The Ingrid-and-Gunther-Schroff Foundation (administered by the Dean of the University of Karlsruhe), the Karlsruher Hochschulvereinigung, and the Institute for Program Structures and Data Organization (Chair Prof. Tichy) awarded several generous travel grants on the occasions of tournament participation between May 1995 and April 1997. A.S.E. GmbH loaned hardware during the 13th WMCCC and the 14th WMCCC for free. Digital Equipment Corp. provided two 500MHz Alpha-21164a computers under external technology research contract DE-0032-97 in March 1997. Sun Microsystems Inc. donated a 167MHz UltraSparc Creator workstation for program development and 64-bit Sparc V9 benchmarking in February 1996.

A Test Suite Results of DarkThought

Test Suite	Time	Result	Speed
BT-2630 (DarkThought rating = 2494)	15 min	27 / 30	264K nps
LCT-II (DarkThought rating = 2535)	10 min	26 / 35	249K nps
Bratko/Kopec	1 min	18 / 24	214K nps
Encyclopedia of Chess Middlegames	20 sec	609 / 879	249K nps
Win at Chess	5 sec	287 / 300	267K nps
World's Greatest Chess Games	10 sec	22 / 50	224K nps
1001 Brilliant Ways to Checkmate	5 sec	946 / 1001	313K nps
1001 Winning Chess Sacrifices	5 sec	830 / 1001	259K nps
Nolot Positions (#1 Nxh6! 20:40 min, #10 Rxf7! 44:25 min, #11 Rxh6! 2:40 min)	60 min	3 / 11	210K nps

Solution Times for BT-2630:

#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	(in sec.)
0	2	6	48	0	0	0	1	–	6	**#00**
1	14	572	2	8	–	9	0	24	2	**#10**
122	2	179	7	12	–	4	119	208	48	**#20**

Solution Times for LCT-II:

#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	(in sec.)
15	0	–	–	66	42	264	–	5	–	**#00**
52	164	525	0	2	0	1	110	3	12	**#10**
11	15	–	15	–	–	0	0	3	30	**#20**
5	2	25	–	–						**#30**

Test Configuration:

- DarkThought as of July 30, 1997, with 4M transposition-table entries (2M per side), 512K King hash-table entries (256K per side), and 256K Pawn hash-table entries,
- Digital Unix 4.0b program development tools and operating system,
- 500MHz Alpha-21164a workstation (500MHz DEC Alpha-21164a EV56 CPU, 8KB/8KB on-chip I/D L1 caches, 96KB unified on-chip L2 cache, 1MB unified off-chip L3 cache, DEC PC164 mainboard, 8x16MB PS/2 SIMMs = 128MB RAM) as provided by Digital Equipment Corp. under external technology research contract DE-0032-97.

B Tournament History of DarkThought

8th World Computer Chess Championship (May 1995):

- finished 14th among 24 participants with 2.0 points out of 5 games (2x win, 3x loss).

13th World Microcomputer Chess Championship (October 1995):

- finished 7th among 34 participants with 7.0 points out of 11 games (6x win, 2x draw, 3x loss) – best European amateur and best German overall.

11th AEGON Man vs. Machine (April 1996):

- 3.5 points out of 6 games (2x win, 3x draw, 1x loss) – sole loss against GM David Bronstein (ELO 2455).

14th World Microcomputer Chess Championship (October 1996):

- finished 7th among 27 participants with 6.5 points out of 11 games (5x win, 3x draw, 3x loss) – ranked 3rd in the speed-chess tournament.

12th AEGON Man vs. Machine (April 1997):

- 3.5 points out of 6 games (2x win, 3x draw, 1x loss) – win against IM Sofia Polgar (ELO 2470), draw with GM Cifuentes Parada (ELO 2539), sole loss against IM Johan van Mil (ELO 2406).

1st Karlsruher UniSchau (June 1997):

- 3.0 points out of 4 games (2x win, 2x draw) – exhibition match against the official student chess team from the University of Karlsruhe (ELO 2225 – ELO 2305).

References

[1] Adelson-Velsky, G. M. and Arlazarov, V. L. and Bitman, A. R. and Zhivotovsky, A. A. and Uskov, A. V. (1969). Programming a computer to play chess. *1st Summer School on Mathematical Programming*, Proceedings (Vol. II), pp. 216–252 [another translation of the Russian original appeared in *Russian Mathematical Surveys*, Vol. 25, pp. 221–262, London Mathematical Society, 1970].

[2] Akl, S. G. and Newborn, M. M. (1977). The principal continuation and the killer heuristic. *ACM '77 National Conference*, Proceedings, pp. 466–473, ACM Press.

[3] Anantharaman, T. S. (1991). Extension heuristics. *ICCA Journal*, Vol. 14, No. 2, pp. 47–65.

[4] Anantharaman, T. S. and Campbell, M. S. and Hsu, F.-H. (1988). Singular extensions: Adding selectivity to brute-force searching. *ICCA Journal*, Vol. 11, No. 4, pp. 135–143.

[5] Beal, D. F. and Smith, M. C. (1996). Multiple probes of transposition tables. *ICCA Journal*, Vol. 19, No. 4, pp. 227–233.

[6] Beal, D. F. and Smith, M. C. (1995). Quantification of search extension benefits. *ICCA Journal*, Vol. 18, No. 4, pp. 205–218.

[7] Beal, D.F. (1989). Experiments with the null move. *Advances in Computer Chess 5*, D. F. Beal (ed.), pp. 65–79, Elsevier Science, ISBN 0-444-87159-4.

[8] Berliner, H. J. and McConnell, C. (1996). B* probability based search. *Artificial Intelligence*, Vol. 86, No. 1, pp. 97–156.

[9] Berliner, H. J. and Ebeling, C. (1990). Hitech. *Computers, Chess, and Cognition*, T. A. Marsland and J. Schaeffer (eds.), pp. 79–109, Springer, ISBN 0-387-97415-6/3-540-97415-6.

[10] Berliner, H. J. (1987). Some innovations introduced by Hitech. *ICCA Journal*, Vol. 10, No. 3, pp. 111–117.

[11] Bettadapur, P. (1986). Influence of ordering on capture search. *ICCA Journal*, Vol. 9, No. 4, pp. 180–188.

[12] Brockington, M. G. (1996). A taxonomy of parallel game-tree search algorithms. *ICCA Journal*, Vol. 19, No. 3, pp. 162–174.

[13] Campbell, M. S. and Marsland, T. A. (1983). A comparison of minimax tree search algorithms. *Artificial Intelligence*, Vol. 20, No. 4, pp. 347–367.

[14] Cracraft, S. M. (1984). Bitmap move generation in chess. *ICCA Journal*, Vol. 7, No. 3, pp. 146–153.

[15] Donninger, C. (1993). Null move and deep search: Selective search heuristics for obtuse chess programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137–143.

[16] Gillogly, J. J. (1972). The technology chess program. *Artificial Intelligence*, Vol. 3, No. 1–3, pp. 145–163 [reprinted in *Computer Chess Compendium*, D. N. L. Levy (ed.), pp. 67–79, Springer, 1989, ISBN 0-387-91331-9].

[17] Goetsch, G. and Campbell, M. S. (1990). Experiments with the null-move heuristic. *Computers, Chess, and Cognition*, T. A. Marsland and J. Schaeffer (eds.), pp. 159–168, Springer, ISBN 0-387-97415-6/3-540-97415-6.

[18] Heinz, E. A. (1998). Efficient interior-node recognition. *ICCA Journal*, Vol. 21, No. 3, pp. 156–167.

[19] Heinz, E. A. (1998). Extended futility pruning. *ICCA Journal*, Vol. 21, No. 2, pp. 75–83.

[20] Hyatt, R. M. and Gower, A. E. and Nelson, H. L. (1990). Cray Blitz. *Computers, Chess, and Cognition*, T. A. Marsland and J. Schaeffer (eds.), pp. 111–130, Springer, ISBN 0-387-97415-6/3-540-97415-6.

[21] Knuth, D. E. and Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326.

[22] Korf, R. E. (1985). Iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, Vol. 27, No. 1, pp. 97–109.

[23] Levy, D. N. L. and Broughton, D. C. and Taylor, M. (1989). The SEX algorithm in computer chess. *ICCA Journal*, Vol. 12, No. 1, pp. 10–21.

[24] Marsland, T. A. (1992). Computer chess and search. *Encyclopedia of Artificial Intelligence*, S. C. Shapiro (ed.), pp. 224–241, Wiley & Sons, 2nd ed. 1992, ISBN 0-471-50305-3.

[25] Marsland, T. A. (1986). A review of game-tree pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19.

[26] Nelson, H. L. (1985). Hash tables in Cray Blitz. *ICCA Journal*, Vol. 8, No. 1, pp. 3–13.

[27] Plaat, A. and Schaeffer, J. and Pijls, W. and de Bruin, A. (1996). Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, Vol. 87, No. 1–2, pp. 255–293.

[28] Reinefeld, A. (1983). An improvement to the Scout tree-search algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4–14.

[29] Schaeffer, J. and Plaat, A. (1996). New advances in alpha-beta searching. *24th ACM Computer Science Conference*, Proceedings, pp. 124-130, ACM Press.

[30] Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212.

[31] Schaeffer, J. (1986). *Experiments in Search and Knowledge*. Ph.D. Thesis, University of Waterloo [reprinted as Technical Report TR 86–12, Department of Computing Science, University of Alberta].

[32] Schaeffer, J. (1983). The history heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19.

[33] Scott, J. J. (1969). A chess-playing program. *Machine Intelligence 4*, B. Meltzer and D. Michie (eds.), pp. 255–265, Edinburgh University Press, 1973, ISBN 0-852-24062-7.

[34] Slate, D. J. (1984). Interior-node score bounds in a brute-force chess program. *ICCA Journal*, Vol. 7, No. 4, pp. 184–192.

[35] Slate, D. J. and Atkin, L. R. (1977). CHESS 4.5 – The Northwestern University chess program. *Chess Skill in Man and Machine*, P. W. Frey (ed.), pp. 82–118, Springer, 2nd ed. 1983, ISBN 0-387-90790-4/3-540-90790-4.

[36] Uiterwijk, J. W. H. M. (1992). Memory efficiency in some heuristics. *ICCA Journal*, Vol. 15, No. 2, pp. 67–74.

[37] Wendroff, B. (1985). Attack detection and move generation on the X-MP/48. *ICCA Journal*, Vol. 8, No. 2, pp. 58–65.

[38] Ye, C. and Marsland, T. A. (1992). Experiments in forward pruning with limited extensions. *ICCA Journal*, Vol. 15, No. 2, pp. 55–66.