Evolution of the Chess Robot: Brute force Wins

Jonathan Babb*
MIT Laboratory for Computer Science
Cambridge, MA 02139

January 10, 1996

Chess masters used to come to computer chess tournaments to laugh. Now they come to watch. Soon they will come to learn. Monty Newborn

Abstract

The idea of an automaton playing chess has been around for over 200 years. Top ranking chess robots now contend with grandmasters. This paper describes the evolution of these robots into brute force game-tree searchers with an insatiable desire for computation power. In particular I compare the anatomies of three modern robots, Hitech (USCF 2400), Deep Thought (USCF 2600), and Startech (estimated USCF 2400) and develop a simple performance model in terms of their relative knowledge and search speed. These machines are capable of approaching the performance of the top human players with knowledge only comparable to the average chess player. The bottom line is that while the knowledge factor is important, machines can overcome their knowledge inferiority through increased search speed alone. I conclude by presenting research direction for an FPGA-based reconfigurable chess engine.

1 Introduction

1.1 The Drosophila of AI

One of the holy grails of computer science has long been to create a machine that can beat the human World Chess Champion. This grand challenge is particularly interesting because of the existence of an established quantitative metric - the Elo rating system [23]. The strength (or weakness) of a given machine can be determined by direct tournament play against human or other machine players. In addition, standard tests [35] exist for reliably measuring the performance of a chess machine. According to John McCarthy, one of the founders of artificial intelligence (AI), chess is in fact the Drosophila[1] of AI [41]. McCarthy indicated that for AI applications to be developed, basic research, such as in the field of computer chess, is essential.

1.2 The Need for Speed

However, chess seems to have fallen from grace as mainstream AI research. Rather than solving the problem using cognitive, knowledge intensive methods similar to humans, chess machines have taken on a more brutal personality, characterized by a focus on enormous search speeds. A linear correlation between computer speed and chess performance [18], 80 – 100 rating points for every doubling of speed, has lead engineers to expend their chief efforts increasing search speeds. For example, a chess machine under development at IBM, Deep Blue, is predicted to have a search speed in excess of 1 billion position evaluations per second [17]. In contrast, humans may only reason about a hundred positions (at most!) before deciding their move [20]. Although some chess knowledge is important for computers, in many cases additional knowledge actually hurts the performance of such a machine because of the accompanying decrease in search speed [30].

Figure 1 (from [11]) shows this knowledge-speed tradeoff. The X axis measures search speed while the Y axis measures program knowledge. Each curve represents a constant rating level achieved with various combinations of knowledge and search strength. For a machine with fixed resources, any increase in knowledge must come at the expense of search speed. As we move from point 1 to point 2 in the diagram, overall performance increases with knowledge. However, at point 3 the search speed is degraded such that the overall performance decreases with additional knowledge. This figure is somewhat oversimplified. In Section 5.2 we will explore the evolution of this tradeoff in winning chess robots.

In addition, while increasing machine knowledge often requires AI breakthroughs, machine speed can generally be increased with the application of known computing principles.

[1] Referring to Thomas Morgan's Nobel prize winning work with fruit flies in 1933.

* Area Exam. January 1996.

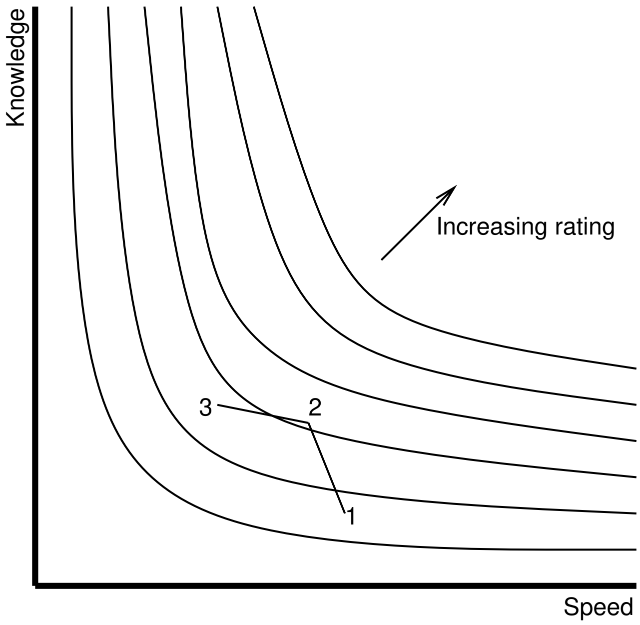

Figure 1: Qualitative Knowledge/Speed Tradeoff

Player	USCF Rating	Speed (nodes/s)
Master	≈ 2200	—
Belle (1982)	≈ 2200	$160K$
Cray Blitz (1983)	≈ 2200	$60K$
Hitech (1987)	≈ 2400	$120K$
StarTech (1994)	≈ 2400	$350K$
Grandmaster	≈ 2600	—
Deep Thought (1989)	≈ 2600	$900K$
World Champion	≈ 2900	—

Table 1: USCF Rating of Chess Machines

1.3 Master Chess Machines

While directing increasing amounts of computational power at searching chess game-trees may ostensibly be straightforward, the efficient utilization of this power is often elusive. This paper will survey some of the more successful attempts, characterized by the master chess machines in Table 1. Many of these machines were computer World Chess Champions in their time, and Table 1 shows their relative United States Chess Federation (USCF) ratings in comparison with human masters and the human World Chess Champion. Table 1 additionally shows the machine's reported search speeds[2].

For computation power, these machines rely on the workhorse of supercomputer processors (Cray Blitz [31]) or the combined parallel processing power of a massively parallel processor (StarTech [38] on a 512 processor CM5 [16]). Alternately, machines may take advantage of hordes of special-purpose hardware (Belle [19] and Hitech [12]), or even parallel instances of special-purpose hardware (DeepThought [50]). For knowledge, these machines rely on the basic alpha-beta search, to be described in Section 3, in conjunction with an arsenal of search enhancements and extensions.

This paper focuses on the anatomy of the following three machines:

- Hitech - A chess machine completed at Carnegie Mellon University (CMU) in 1985 consisting of a special-purpose 64-chip custom VLSI move generator as well as hardware for move evaluation and search control.
- DeepThought - A chess machine completed at CMU in 1988 consisting of a single chip custom VLSI move generator and hardware for move evaluation and search control, of which multiple instances may be run in parallel.
- StarTech - An implementation of the Hitech algorithm with a parallel Scout [45] search algorithm on a 512-node CM5 completed at the Massachusetts Institute of Technology in 1993.

This paper explains how these chess machines achieve their remarkable performance. For background, Section 2 provides a short record of significant computer chess events. Section 3 then describes the alpha-beta search algorithm and presents the various enhancements that chess algorithms have employed along the way. I detail the architectural features of chess robots in Section 4, comparing how different chess machines apply brute force speed to achieve master performance. In Section 5.2, I present and apply a simple performance model for comparing the knowledge and search speed of chess robots. Section 6 presents a proposal for constructing a chess machine from reconfigurable logic and explains the merits and faults of such a system. I make concluding remarks in Section 7.

2 Background

Although chess is one of many two player games, its status as the ultimate test of intellectual skill has merited its own field of research. To place proper perspective on the master robots I am examining, I first briefly trace the history of this challenging (and fun!) field.

[2]Note that due to varying knowledge, the search speed does not directly correspond to rating. StarTech's speed was estimated from a speedup of 50 on 512 nodes with a sequential speed of $7K$ nodes/second.

2.1 Chronology of Computer Chess

1770: Turk - The first reported chess robot, "Turk", was constructed in 1770 by Hungarian Baron Wolfgang von Kempelen [10] [17]. The automaton was counterfeit - a small chessplayer was ingeniously buried amongst a rather large amount of clockwork and accounted for the robot's apparent success - including supposedly beating Napoleon. It toured Europe and Edgar Allan Poe revealed its secret in an essay.

1890: Torres' Chess Machine - In 1890, Spanish Engineer Torres y Quevedo invented an actual machine that played out the king and rook versus king endgame [52]. This electromechanical robot played the king and rook side and used six programmed rules to determine its response to any defense adopted by its opponent.

1950: Classic Shannon paper - In 1950 Claude E. Shannon of Bell Telephone Laboratories published the seminal "Programming a Computer for Playing Chess" [48], which elaborated the basic principles of computer chess. The first important computer chess paper, his work estimate the number of possible chess positions (10^{43}) and suggested applying a minimax algorithm based on an approximate static evaluation function. He identified three possible types of program play:

- Type A: Full search to a fixed depth.
- Type B: Selective searching via pruning.
- Type C: A goal oriented procedure.

He also recommended the use of differing strategies for opening, middle, and end game phases, as well as recognizing the need for a quiescent search (see Section 3.4.1). He suggested that the computer should make more use of brutal calculations than humans, but that a little knowledge and selection might go a long way.

1953: Classic Turing paper - Independent of Shannon's work, Turing [51] published a paper describing how a computer might play chess and presented a hand simulated program. He entertained the possibility (to the dismay of many philosophers) of programming a computer to play chess better than the programmer.

1956: Los Alamos Type A Program - Ulam and Stein worked on a Type A program at Los Alamos, N.M. [42]. On a reduced 6x6 board, a 600 word program was capable of a full search to a depth of 2 in 12 minutes of MANIAC-I time.

1958: IBM Type B Program - Alex Berstein headed development of a Type B program at IBM [42]. Running on an IBM 704, this 7000 word program played actual chess, searching to a depth of 2 in 8 to 12 minutes by considering at most 7 branches at each move.

1959: Rand Type C Program - Newell, Simmon, and Shaw tackled a more cognitive Type C program at Rand Corporation [42]. Their program, NSS, running on the JOHNNIAC required from one to ten hours per move and could supposedly play at the 1100 USCF level [17].

1966: Stanford versus Moscow - In 1966 a chess playing program which John McCarthy migrated from MIT (based on Kotok's 1962 Bachelor's Thesis [36]) lost a four game match to a program developed at the Institute of Theoretical and Experimental Physics in Moscow. The Stanford program ran on an IBM 7090 and the Soviet program on a Soviet M-20 computer.

1966: The Greenblatt Chess Program - Greenblatt's program $MacHack$ 6 [28] was the first program to compete at the level of the average U.S. tournament player. Developed at the Artificial Intelligence Laboratory of Project MAC at MIT, the program employed book openings, alpha-beta search, and a hash table for avoiding consideration of a position twice. $MacHack$ 6 established a performance rating above 1600 USCF. Later, Greenblatt attempted to improve his program with the addition of the CHARM hardware, capable of searching 150K move/sec, but the minimal knowledge in this fast search did not prove successful.

1972: The Technology Chess Program - Gillogy developed the Technology program (TECH) [26] as an increasing baseline for chess programming against which more sophisticated programs could compare. TECH was a brute force search with material evaluation only whose performance would continue to improve as computer technology increased.

1970-1978: Northwestern University CHESS 3.0-4.9 - In 1970 Monty Newborn organized the first United States Computer Chess Championship, and in 1974 a World Computer Championship was established. With the exception of a 1974 lost to the Soviet program KAISSA [2] (with its Method of Analogies), the Northwestern Universities program CHESS 3.0 to CHESS 4.9 [49] was the leading competitor, achieving a rating near 2100 USCF by 1978. This program performed a fixed-width search to a specified depth and was the first to exhibit an undeniable increase in performance as the program was migrated to faster machines.

1979 - 1982: Belle - Belle, developed by Ken Thompson (co-creator of Unix) and Joe Condon [19], was the first program to reach the Master level (2200 USCF) and relied on the speed of special-purpose hardware that allowed it to exhaustively search over 8-ply deep in the middle game. With a vast opening library, Belle dominated the computer chess scene until 1983.

1983: Cray Blitz - Without special hardware (albeit it did run on the world's fastest computer in 1983, the Cray XMP), the program Cray Blitz was champion in 1983, searching 3 million moves in 3 minutes during the middlegame. On the multiprocessor Cray YMP-8, this program was later able to achieve search speeds near 200K positions per second. Blitz employed a considerable amount of chess knowledge compared to Belle and other hardware implementations, and was the first to take on a parallel search.

1985: Hitech - With custom VLSI for move generation, special hardware for move evaluation, and a host program that performs "Oracle" analysis, the Hitech [12] machine completed at CMU in 1985 immediately became the world's best computer chess machine, eventually earning a rating of nearly 2400 USCF.

1987: Microcomputers become Masters - Targeting the personal computer market and average chess players, microcomputer programs such as Sargon, Chess Challenger, Socrates, Mephisto, and Chessmaster became popular. In 1987 the program Fidelity passed the 2200 USCF mark, and Mephisto followed in 1988.

1988: Deep Thought - With a single chip legal move generator, Deep Thought was the first chess machine to achieve Grandmaster status (2600 USCF). Developed at CMU, Deep Thought and its descendant Deep Thought II immediately became the de-facto champion machine.

1995: 8th World Computer Chess Championship - Chess program Fritz, running on a 90MHZ Pentium PC, becomes World Computer Chess Champion, demonstrating that chess knowledge was at least as important as computing power. Fritz defeated the favored Deep Thought II (called Deep Blue Prototype in the tournament) in the playoff versus StarSocrates [32] from MIT. StarSocrates employs the same parallel search algorithm as StarTech and was running on a 1824 processor Intel Paragon.

1996: Deep Blue versus Kasparov - Under develop at IBM, Deep Blue [17] is a 1,000 processor machine based on a modern implementation of the Deep Thought processor. In February 1996 Deep Blue will match human World Champion Gary Kasparov to contend for the World Chess Championship.

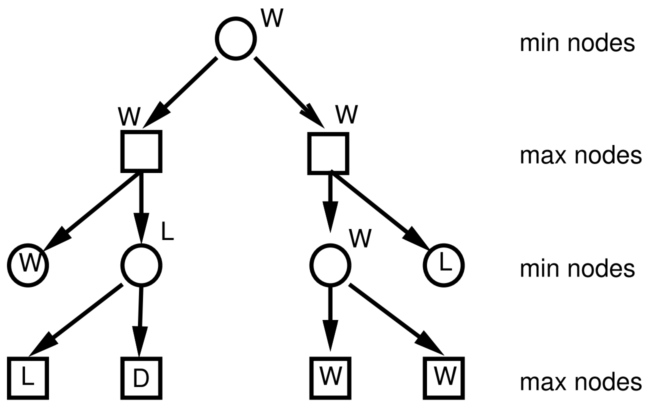

Figure 2: An Evaluated Game Tree

3 Intelligent Game Tree Search

Chess, like tic-tac-toe, checkers, kalah, and GO, is a two-person, perfect-information game. Adversary players take turns making decisions (moves) allowed by the rules of the game. A *game tree* explicitly enumerates all potential plays from the initial position and ends in leaf positions which are always a win, loss, or draw. A particular game is represented by a path from the initial position (the root of the tree) to a leaf node. Figure 2 shows a simple 3 level game tree with the intermediate nodes (circles for the first player and squares for the second) annotated with the solution as determined by the minimax algorithm to be presented in Section 3.3.

3.1 Solving Chess

Before delving into the details of the game tree search, let us discourse, for a moment, on the possibility of completely solving the game of chess. I define a solution of chess as follows:

- A solution to chess is an algorithm and/or database capable of playing perfect chess from the initial position under tournament time constraints.

If the solution entails enumeration of all of the supposedly possible 10^{120} games (the Shannon number), then statements such as Patrick Winston's "if all the atoms in the universe had been computing chess moves at picosecond speeds since the big bang (if any), the analysis would be just getting started" [53] tend to overwhelm our imagination. However, let us examine some more relevant research.

3.1.1 How Many Positions?

First, let's more closely examine Shannon's 1950 numbers [48]. In chess, the game tree nodes do not necessarily correspond to unique positions. That is, a position may be encountered through more than one combination of legal moves.

This line of thought leads to another often quoted number of Shannon's - his estimate of the number of legal board positions. He arrived at $64!/32!8!^2 2!^6 \approx 10^{43} \approx 2^{142}$, which is actually just the total number of combinations (legal or not) of all the pieces on an 8 X 8 board. This number does not consider other combinations of pieces (of which there are 486 per side), but it does count a large number of illegal (broken), and otherwise uninteresting positions. More recent research [43] states that only 136 bits are necessary to store any *legal* position (implying there are at most 2^{136} total positions). However, even this number seems much to high for the number of unbroken positions. In addition, while a position may be statically legal, it may not be legally arrivable from the initial position. For example, certain combinations of pawns, pawns on your on back row, and bishops on the same color are not possible.

In 1965 De Groot [20] reported on the ability of chess masters to accurately reconstruct realistic chess positions from memory in comparison with a high recollection error rate for random piece placement. This research tends to support the fact that besides broken positions, the typical random positions may not be of interest in solving chess.

3.1.2 Information Content of Chess Positions

One helpful estimate of the number of interesting positions, or *master positions*, can be obtained from considering the information content of a typical chess master game position. Nievergelt [43] used a series of yes/no questions to determine that the average number of bits requires to guess realistic master positions is 70. Balkenhol [21] used a similar technique to construct 2500 positions in 79 bits on average. An alternate data compression technique [4] for moves (not positions) used Hufmann encoding and an actual chess algorithm for ordering the possible moves. They reduced storage by 25 percent by taking advantage of the fact that not all legal moves in a position have the same probability to be played by a good player.

3.1.3 A Correctional Database

The move ordering approach to storing chess positions leads to an interesting database storage alternative not considered in the chess literature. By taking advantage of a good (and deterministic) chess playing algorithm, a chess solution need not store the answering move for every position, but only corrections for positions in which the algorithm does not choose a winning move. In observed chess play, I notice that the likelyhood of a fatal mistake decreased as a players material advantage increases, so I speculate that a database that only holds the complementary corrections for near-even games may generate a near perfect player. A rare number of positions are equal - while there are 2^{18} total combinations of pieces, only 2^{13} of these combinations have equal material[3]. In addition, not all positions that are even are *evenly arrivable* from the initial position. For example an even game where each player has a made an obvious mistake is not evenly arrivable. Whether or not this approach is feasible will require additional research.

3.2 Are We There Yet?

Quantifying these results in terms of the actual number of chess positions that need to be considered is not easy. However they tend to support the fact that the amount of information and time required to solve chess is much lower than initial estimates. As we shall see in the following sections, effective search pruning techniques similarly reduce the size of the game tree that an algorithm must consider. Whether or not the combination of these reductions is enough for the solution of chess to be surmountable with less than all the atoms in the universe remains to be seen.

As a final note to this section, I point out that all 5 piece and most 6 piece chess endgames are now solved [39].

3.3 Minimax Search with Static Evaluation

Assuming that we can not yet entirely solve the game of chess, what are we to do? Most chess programs use some variant of the *minimax* tree search to evaluate an unsolved chess position. This basic search technique is first described by von Neumann and Morgenstern and outlined as the basis for a chess program in Shannon's paper as described in Section 2. Since the entire game tree can not be solved, the legal moves of the game are generated to a certain set of leaf positions. The value of each terminal position must be estimated with a static evaluation function which does not consider further play. Then, by assuming that each player makes the best move at each level of the game tree, and that a positive value indicates advantage to one player and a negative value advantage to the other, a value for each position can be determined by backing up from the leaves, alternately minimizing and maximizing the score at each level. Thus the minimax algorithm is a look-ahead procedure. In Figure 2, each node has two branches. More typically for chess middlegames, each node has an average of about 35 branches, leading to a combinatorial explosion of positions and the so called Shannon number of nodes for the full game tree.

One advantage of minimax is that it guarantees perfect play - a winning position will always be won and a drawn position will at least be drawn. A disadvantage of the minimax procedure is that the entire board valuation is minimaxed into a single static value that tells little about how it was determined. Berliner and Palay add a confidence factor to this value with the B* Probability Based Search [14] [44], but as yet B* is not

[3] I assume the value of the pieces as follows: $Q = 9, R = 5, B = 3, KN = 3, P = 1$.

incorporated in any champion chess playing machines. The following section demonstrates how the minimax algorithm works in more detail along with alpha-beta pruning.

3.3.1 The alpha-beta Algorithm

The α-β algorithm speeds up game tree searching without loss of perfect play by pruning subtrees that do not need to be searched. The algorithm determines the minimax value by traversing the game tree, but avoids nodes that can no longer influence the minimax value at the root. Winston [53] and Pearl [46] both provide excellent tutorials for this technique while [34] provides the first in-depth analysis. The net results are that two new parameters are added to the recursive minimax algorithm as follows:

- α-bound: The cutoff bound for a MIN node J is a lower bound called α equal to the highest current value of all MAX ancestors of J. The exploration of J can be terminated as soon as its current value equals or falls below α.
- β-bound: The cutoff bound for a MAX node J is an upper bound called β equal to the lowest current value of all MIN ancestors of J. The exploration of J can be terminated as soon as its current value equals or rises above β.

Simplified pseudo-code for this algorithm in negamax notation (where one player's evaluation is the negation of the other's) as presented by Kuszmaul [37] is as follows:

```
Define ab(n,a,B) as
 If n is leaf return eval(n).
 Let C = the children of n, and
   b = -infinity.
  For all c in C do:
   Let s = -ab(c,-B,-a).
    If s >= B return s.
    If s >  a then a = s.
    If s >  b then b = s.
  enddo
 return b.
```

where n is an initial node to be searched and the initial values are $\alpha = -\inf, \beta = +\inf$. These initial values are called the *search window* and are used in some of the enhancements I will explore in the next section.

3.3.2 Move Ordering

The alpha-beta algorithm works best when the game tree is searched in an order such that better moves come first. In the optimal case the game tree is best-ordered and the alpha-beta algorithm searches the minimum number of nodes possible to value the root positions [34]. This happens simply because the bounds, determined by the best alternative searched so far for each side, are tighter if the best moves are found first. This best line of play is called the *principle variation*. In a best-first order game tree of height h and branching degree d, the alpha-beta algorithm is optimal and will evaluate (for even h): $2d^{h/2} - 1$ node, allowing the algorithm to search about twice as deep as the equivalent minimax procedure. Thus alpha-beta reduces the rate of the combinatorial explosion, but does not prevent it.

In practice chess game trees are not optimally ordered, however Kuszmaul reports ordering heuristics capable of achieving 85 – 95 percent best-ordered trees [37] for StarTech.

3.3.3 Game Tree Pathology

As a final note on minimax, I observe that the general assumption that increasing search depth improves play is not an obvious one. Probabilisticly, searching deeper merges together more erroneous static evaluations and should decrease the information content of the valuation. In practice this phenomenon, called game tree pathology [1], is not observed in computer chess, possibly due to dependent values of sibling nodes and the non-uniform nature of the chess game tree. Thus the deeper the search, the higher the quality of play.

3.4 Enhancements

In the course of chess history, many enhancements to the basic alpha-beta search have been incorporated into chess robots. Most, but not all, of the these optimizations are search speedups and do not change the quality of the search itself.

3.4.1 Quiescent Search

Both Shannon and Turing recognized the problem of statically evaluating terminal positions which are not "quiet". For example, in the process of a queen trade we should not stop the search while one side is a queen ahead. The opponent would be able to re-capture, but the evaluation might not easily take this into account. In the Greenblatt program [28], this problem was overcome by adding a secondary search after the initial full-width search. This quiescent search only considered captures, re-captures, and possibly checking moves until all terminal positions are devoid of tactical features. That is, the alpha-beta search continues as usual, with the exception of termination at non-capturing, non-checking moves. Extremely long tactical lines of playing may still be terminated, but at a much deeper search depth than the original fixed-depth search.

All modern chess machines employ some form of quiescent search. Because this search occurs at the leaf nodes of the game tree, the amount of search effort spent in the quiescent phase is typically 20-80 percent [29] of the entire search effort.

3.4.2 Transposition and Hash Tables

An obvious improvement to the search is to use a table to avoid searching the same position twice. Such a table is termed a *transposition* table because it re-uses cached positional information encountered when the same position is played via an alternate ordering of moves. The Greenblatt Chess Program initially employed a hash table of 32K entries; Cray Blitz and StarTech tables use as large as 2^{10} and 2^{30} entries respectively. Each entry represents a board position. Zobrist [54] developed a standard hash table hash mechanism based on the bitwise-xor of the hash codes of a set of properties that are true of a given chess position. Example properties are the position of the king, or the number of pawns. An important component of the transposition table is the replacement policy, which decides when a new entry should override an existing entry - although if the transposition table is large enough the replacement policy is not very important [37].

Transposition tables are most effective in the endgame, where there are fewer irreversible moves, with gains of a factor or 5 or more [49], although middlegame improvements of 25-50 percent have been observed [40]. Interestingly, while Belle and Hitech both included hardware tables, Deep Thought did not use a transposition table for the hardware portion of its search. They claimed only a 10 percent reduction in search speed as a consequent of not hashing the bottom 3-4 levels of the search [30].

3.4.3 Killer Move and History Heuristic

The killer move heuristic is based on the supposition that if move My refutes move Mx, it is more likely that My (the killer move) will be effective in other positions [28]. At nodes of the same depth we re-order the killer moves (trying them first) in an attempt to produce a quick cutoff. While the use of the killer heuristic was not reported to be valuable for TECH, it became a standard optimization for programs such as Chess 4.X and Cray Blitz.

A more general version of the killer heuristic is the history heuristic introduced by Schaeffer [47]. Information about the usefulness of the moves in the previous part of the search is stored in a history table and used to dynamically order the moves at all interior nodes.

3.4.4 Iterative Deepening

In chess tournaments, play is in real time, governed by fixed time limits. Because of the dynamic nature of the search process, it is not possible in advance to predict how long a search to a fixed depth will take. The solution is to start searching shallow and progressively search to deeper levels. Because of the exponential growth of the game tree, the proportion of time spent in the shallower searches is not significant. Basically, new iterations are tried until time is up.

In addition, a $N - 1$ level search can provide valuable information for the Nth level, filling the transposition table and killer lists with useful moves. The literature is inconclusive on the overall benefit of interactive deepening beyond time control, with the exception on Kuszmaul [37] who shows a definite advantage for StarTech due to the improved move ordering for the parallel search.

3.4.5 Aspiration Search

While the previous alpha-beta enhancements focused on improving move ordering, an aspiration search uses a narrower initial search window to improve the algorithm's performance. Given an estimated value V (possibly from a previous, shallower search) and an error limit e, an initial search window of ($\alpha = V - e$, $\beta = V + e$) is tried. If the error limits are hit, the window must be expanded (by increasing e) and the tree re-searched. This technique may be applied recursively, and in the extreme a zero-window ($e = 0$) search may be effective. Note that a zero-window search is equivalent to the scout algorithm [45] that is parallelized in StarTech. In TECH [26] the aspiration search reduced search times by an average of 23 percent.

3.4.6 Null Move Heuristic

With the exception of the quiescent search, all heuristics described up to this point have improved search speed without introducing error. The null move heuristic is a form of forward-pruning algorithm that reduces search effort with a small chance of error. At a particular search level, a player makes a null move (no change in position) that is almost always worse that the best move. That is, the player whose turn it is "passes" and no change is made to the state of the board. In chess this move is not even legal, however the returned value can serve as a lower value on the value of the current position. The main advantage is reduced search depth by one level if the null move yields a cutoff. In rare cases, termed *zugzwang* in chess lingo, a potential null move might actually be a players best move. Null move algorithms are frequently turned off for positions, such as king-pawn endgames, where zugzwang is likely to occur.

The Kaisssa program [2] was the first to use the null move, and Beal [9] proposed a null move quiescent search program. Experiments with the null move in Hitech [27] found a 25 percent reduction in search effort in opening and middlegame positions for Hitech. Deep Thought used the null move, and Kuszmaul estimated that its lack of inclusion in Startech reduced performance by as much as a factor of two.

3.4.7 Selective Search Extension

In Shannon's type B strategy, an algorithm investigates only the most promising lines of play. Besides quiescent searchs, which are a form of selective search, initial efforts were not successful. However, combined techniques used in Deep Thought resulted in an estimated rating increase of 89 USCF. Rather than using heuristic knowledge to prune certain less promising lines of play, in Deep Thought's selective search heuristics are used to search promising lines of play more deeply. The top three extensions reported by Anantharaman in his thesis [5] are:

1. Threat extension (59 USCF points) - extends the search one level into parts of the tree likely to be refuted by deeper searches.
2. singular extension (9 USCF points) - extends single moves that are better than all alternative moves by a significant margin.
3. PV lookahead extension (6 USCF points) - extends moves along the principle variation.

3.5 Parallel Search Techniques

The best way to parallelize[4] the alpha-beta search is not known. It is well known that the problem is a difficult one [25]. Speedup is generally less than the number of processors due to communication and synchronization overhead between the processors as they dynamically expand a highly irregular game-tree. In addition, the parallel implementation generally performs more work (search overhead) than the sequential counterpart.

3.5.1 Parallel Aspiration Search

The earliest documented attempt at parallel search is Baudet's doctoral work in 1978 [8]. In parallel aspiration search the alpha-beta window is partitioned into a number of nonoverlapping subintervals and each interval is solved by a different processor. However a maximum speedup of about 5 or 6, regardless of the number of processors, is a serious limitation.

3.5.2 Tree Splitting

Successful algorithms employ some form of tree-splitting algorithm. Basically, the game tree is split and its nodes allocated to separate processors which perform their own search. Marsland and Campbell [40] explore variations of tree decomposition algorithms including the principle variation (PV) splitting algorithm. Cray Blitz employed an extension called Dynamic Tree Splitting for the 8 processor YMP implementation.

The biggest challenge with tree splitting is avoiding speculative work - not searching nodes that the sequential algorithm would have pruned. If this search overhead is ignored, speedups are limited to at most $\sqrt{P}$ on P processors. Algorithms such as *mandatory work first* [3] and the *Young Brothers Wait Concept* [24] employed by the ZugZwang chess machine lead to low search overheads for well ordered trees. A more comprehensive overview of tree-splitting approaches is given in Ciancarini [15].

3.5.3 Jamboree Search: Parallel Scout

In StarTech [38] and StarSocrates [32], a new tree-splitting search based on a parallel implementation of Pearl's Scout [45] algorithm is presented. Scout is parallelized by first performing the search on the best descendant node, and then performing the zero window searches of the remaining children in parallel. If any zero-window "tests" fail, those children must then be searched with a full window. Search overhead, is reported to be a factor of 2 to 3. With a scheduler to dynamically assign tasks to processors, StarSocrates claims to yield an average available parallelism of over 1000 for tournament chess.

4 Chess Robot Anatomy

The previous section carefully explained the theory behind successful algorithms for searching the chess game tree. This section focuses on the architectural components of chess robots implementing the search. The main computational components of a chess robot, whether implemented in software or hardware, are divided into the following categories:

- move generation,
- position evaluation,
- and search control.

In the following sections I will examine the each component in detail.

4.1 Move Generation and Ordering

The move generator enumerates the set of legal moves in a given position. Because of the importance of move ordering to the alpha-beta algorithm, a high performance move generator must also heuristically order the moves such that good moves are more likely to come first. In addition, the ideal move generator should not produce all the positions at once, since due to search cutoff a significant fraction of the moves at each position may not need generating at all.

In sequential, software-only implementations, move generation typically accounts for 40 to 50 percent of the processor's time (although in TECH this ratio sky-rocketed to over

[4] Here I refer to coarse-grain parallelization of the search itself, and not fine-grain parallelization of the inner-loop (move generation and evaluation) such as employed in hardware chess machines.

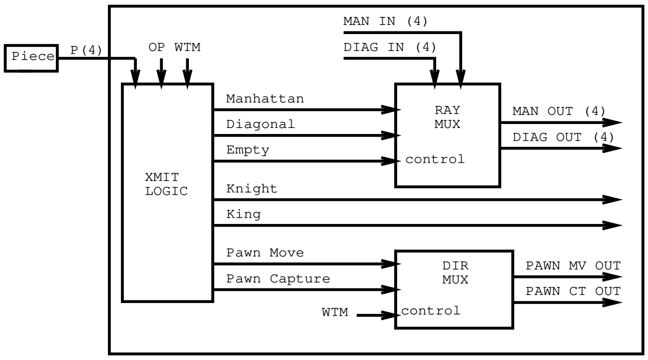

Figure 3: Deep Thought Move Generation Transmitter

90 percent due to the over-simplified evaluation function) . Because of the difficulties of generating the moves in the appropriate order, these implementations tend to produce and then sort all moves. The best software-only robots (including StarTech) tend to have hand-assembled move generators. Cray Blitz takes advantage of the bit-level parallelism of its 64-bit processor by using special 64-bit *bit-boards*.

4.1.1 Belle and Deep Thought Design

The first successful hardware move generator was incorporated in the Belle design, and an improved single-chip implementation of Belle's design is used in the Deep Thought progeny [29]. This design is based on refinements of an early proposal by Babaoglu [6] and uses the *most valuable victim / least valuable aggressor* move ordering heuristic. Basically this heuristic chooses the untried move which captures the largest valued opponent piece (the victim) with the smallest valued friendly piece (the aggressor).

This move generator is arranged in an 8 x 8 array of combinatorial circuits which perform two operations - FIND-VICTIM and FIND-AGGRESSOR. In addition, each array element has a 4-bit piece register containing the value of the piece on the corresponding square. The logic of each element can be further divided into a transmitter and a receiver. Figure 3 shows the implementation of the transmitter logic used in Deep Thought. During the FIND-VICTIM cycle the transmitters generates attacks radiating from each square based on the piece on that square. The outputs of the receivers under attack are then arbitrated (via an arbitration network in Deep Thought and a priority network in Belle) to determine the largest victim. In the FIND-AGGRESSOR cycle this victim then transmits as the union of all pieces (the super-piece) and the same network is used to find the least aggressor. Additional logic is used for special chess moves.

4.1.2 Hitech Design

Hitech uses a very different approach for move generation. An 8 x 8 array of circuits is again used, but in this design each

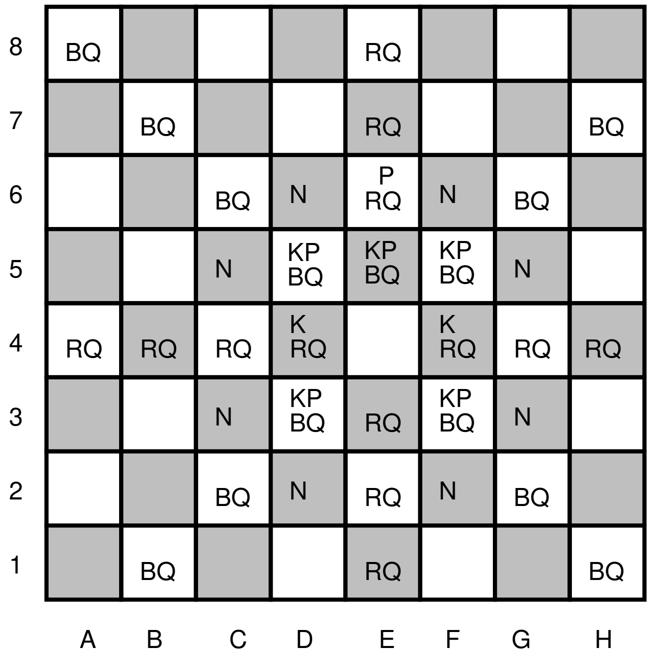

Figure 4: Ever-Possible Moves to the E4 square

circuit determines which moves out of the "ever-possible" moves are legal. There are about 4000 ever-possible moves, described by the set of triples (piece, origin square, destination square) which are legal in the game of chess. Figure 4 shows the set of possible moves to the E4 square. In Hitech each square's circuitry is on a separate chip and the entire board state is distributed to each chip. The job of E4's circuit is then to determine which of its ever-possible moves are legal for the given board state. A hardware priority encoder similar to that in the Belle design is used to order the resulting legal moves. Hitech's move ordering is similar to Belle's with the addition of a safety factor. However simulations have shown this factor to actual reduce the move ordering efficiency.

4.1.3 Comparison

In comparison to the Deep Thought chip, the Hitech move generator is clearly an inefficient design and does not merit further exploration. In fact, the Deep Thought design corrected most of the major flaws in the original Belle design and is a very efficient design, only requiring about 35K transistors. The move generator was a good candidate for hardware implementation because of the following factors:

1. Inner loop: Move generation is a time consuming inner loop of the search algorithm.
2. Fine-grain parallelism: Each legal move is computed in parallel.
3. Bit-level operations: Computations involve small bit-level operations.

4. Specialization: Each chess square gets its own circuit. A hardware arbitration network is efficient.

5. Hardwired communication: Communication between chess squares is hardwired rather than requiring memory bandwidth.

6. Low interface bandwidth: 30 bits per legal move generated (two 10-bit writes and a returned 10-bit value).

While there are surely improvements to the Deep Thought design, they are not obvious and one would tend to consider this component of the robot anatomy a solved problem for a single processor machine.

4.2 Position Evaluation

Ideally, the search process would reach terminal positions with values of win, lose, or draw. However, as explained in Section 3, the search will stop at a certain non-final positions and the algorithm is required to statically evaluate the position to estimate the probable outcome of a complete search from that position. The value returned generally takes into account both material and positional advantage. Material advantage is simple and can be computed incrementally after each move. Positional advantage is not so easy and the evaluation of positional features encompasses the majority of the knowledge in a chess robot.

4.2.1 Incremental Evaluation

In Ebeling's Hitech thesis [22] the complexity of evaluation functions is classified into three categories:

- First-order (Incremental)- function is a linear sum of the values of each square where the value of a square is a function of only the piece on that square.
- Second-order (Slow) - function depends on the relationship between two or more squares.
- Higher-order - function depends on game history (generally not used).

The first category corresponds to evaluation which is computed incrementally in Deep Thought, Belle, and software implementations as early as TECH. Material advantage is a first-order function. Since each move involves at most three piece changes (two in non-capturing moves), all first-order evaluations can be computed incrementally with one addition and two subtractions. Furthermore, a single 1K word lookup table (size = pieces X squares) can be used to encode any number of first-order functions. As an example consider the first move E2-E4. Assuming that the function $F(0)$ contains the first-order valuation for the starting position and $T(piece, square)$ is the lookup table, we can compute the value of this new position with the following operations: $F(1) = F(0) - T(pawn, E2) + T(pawn, E4)$.

Incremental evaluation is fast. Adding knowledge to incremental valuation, by improving the lookup table weighting, does not affect computation time. The power of the incremental function can be further extracted with the following enhancements:

- Skipping further evaluation: Upon receiving the results of the incremental evaluation the search algorithm may not need further evaluation to cutoff the node.
- Approximation of 2nd-order functions: The incremental evaluator can be used to approximate 2nd-order functions, such as king safety, without any overhead.
- Automatic Tuning: Because the incremental evaluation is simply a series of weights in the lookup table, these weights can be automatically tuned against master play.
- Pre-evaluation: As the game proceeds, the table weights can be changed to weights optimized for positions more like the root position of the search.

If move generation is implemented in hardware, then inclusion of a hardware incremental evaluation function is benefitial. Getting the maximum benefit of the incremental evaluation is an area for further research. While both Deep Thought and Hitech include pre-evaluators to determine the incremental weighting, these weightings are only changed at the beginning of a search. Ebeling mentions the possibility of changing the weights during the search, but states that there is a problem minimaxing values from different weightings. However, after only a few levels of searching the amortized cost of changing the weights is small and should be investigated.

While Hitech does not report automatic tuning, Deep Thought's results are promising. However, the question of what to tune against will become more of an issue as computer strength increases. Once computers outperform all humans, then optimizing the computer to make "master moves" may not be such a good idea. In his thesis, Anantharaman [5] discusses an alternate method of tuning against a deeper search of the same position. A possible direction not explored in the literature is to tune against a *more knowledgeable* search of the same depth.

4.2.2 Slow Evaluation

Because first-order valuation is so fast, second-order evaluation functions are called slow evaluation. While some amount of slow evaluation appears to be valuable, the optimal amount is not known. In software-only machines, slow evaluation tends to be an ad hoc set of rules and attempts to ingrain the robot with "expert" knowledge, possibly through consultation with human masters. The value of a particular chunk of knowledge can be experimentally measured. The overall benefit of typical machine knowledge over material-only knowledge is equivalent to at least two orders of magnitude in search speed.

In Hitech, the slow evaluation function consists of 14 pattern recognizers constructed in a manner similar to its move generator. These recognizers have three levels of 8Kx8 memories to each reduce a total of 36 bits of state information to eight bit values. Context (the result from other recognizers) is also allowed as input. At the beginning of a search these recognizers can be dynamically re-programmed to look for relevant features. However, the literature is not clear in describing what algorithms are used to program the recognizers.

In Deep Thought, the slow evaluation is much simpler, consisting of an 8-way pipelined set of tables for evaluating the pawn structure and rook placement. The results are quickly summed together with a 100M additions/second pipelined adder tree. With this evaluator, isolated pawns, passed pawns, backward pawns, and doubled pawns are recognized as well as the number of rooks occupying each file.

Should slow evaluation be implemented in hardware? Although not as critical as move generation and incremental evaluation, a software slow evaluation would be in the critical path of Deep Thought or Hitech. The real issue is how complex the slow evaluator should be. Both Hitech's and Deep Thought's slow evaluators are limited by what the designers could implement in hardware. Hsu [30] claims that missing space and mobility metrics in Deep Thought cost almost a full ply of extra search. In Hitech, over half of the recognizers would be consumed just to detect any passed pawns.

An interesting point to note as a precursor to Section 6 is that both slow evaluators used Xilinx LCAs in their construction. Furthermore, in Ebeling's thesis he recommends using a dynamically programmable PLA in the paper design of a hypothetical Hitech evaluation chip.

4.3 Search Control and Search Support

Search control coordinates the effort between the brawn in the move generator and the brains in the position evaluator while executing the alpha-beta algorithm. In addition, the search control keeps track of the moves made to get to the current position and implements any search enhancements. Beyond the alpha-beta algorithm itself, the implementation of search control in a sequential software-only chess robot is not particularly interesting. However, it deserves a closer look both for hardware robots and for parallel software.

4.3.1 Hardware Search Control

To take full advantage of hardware move generation and evaluation, the search control of Deep Thought and Hitech must also be in hardware. Both designs use a simple microcontroller to coordinate moving and evaluating as well as to execute the search for the bottom levels of the tree. Because the value of a position is completely encoded in a single number, that value (along with the alpha-beta bounds) is the only information that needs to be passed to the software host. At the bottom of the tree the search control itself is still part of the machine's inner loop. However, after only a few levels, the search of the remaining nodes can be switched to software without significant slowdown. Thus any higher level functionality, such as Hitech's Oracle, can execute on the host.

4.3.2 Parallel Search Control

The search control task is dramatically more complicated for parallel searching. Hitech is not parallel. Although Deep Thought searches with a few processors in parallel, the exact details of the algorithm are not described. For StarTech, the parallel search control is further divided into two components:

1. the Jamboree Parallel Scout Algorithm,
2. and the Task Scheduler.

The Jamboree algorithm, discussed in Section 3.5.3, is executed instead of the regular alpha-beta search. As the Scout algorithm explores each node, separate processes are generated for each zero-window search. The process scheduler is then responsible for allocating these task to the processors of the CM5. Startech uses a work-stealing approach that avoids overloading processors already doing useful computation. This scheduler is reported to achieve actual performance of:

$$T \approx 1.02W/P + 1.5C \tag{1}$$

where T is runtime, P is the number of processors, and W and C are the the total work and critical path of the search. In profiling Startech, Kuszmaul found that roughly 1/3 of the time was spent in communication and synchronization and 1/3 of the time was spent in the control flow of the Jamboree algorithm. The remaining 1/3 of the time was spent on actual move generation and evaluation related processing.

4.3.3 Transposition Table

While implementation of many of the search enhancements is not a problem as we apply special hardware or parallel processing to the task, keeping the transposition table in working order is not trivial. In Hitech, a special hardware transposition table is added to the robot's datapath, similar to the Belle table. This 2^{20} entry table is reported to reduce the search size by a factor of two. In Deep Thought a transposition table is not implemented in hardware. However, since only the bottom 3-4 levels are searched by hardware, a software transposition table can be used for the upper levels to mitigate the loss.

In Startech the challenge is to implement a global transposition table. The transposition table is distributed across all processors and the low-overhead CM5 message passing protocol is used to access it. A portion of the hashing key is used to determine with which processor to communicate. Parallel implementations of Cray Blitz also used a similar global transposition table accessed via a single global memory.

5 Performance Trends

5.1 The Rating System

The standard metric for determining the performance of chess players is to play out a series of games and record the win, draw, or loss outcome of each game. While a win is always worth one point and a draw is always worth a half of a point, the effect of these points on a rating is also determined by the opponents rating. For a match between a player with an established rating r_1 and another established player with rating r_2, the new r_1 rating is set to:

$$r_1^{new} = r_1 + K\left(w - \frac{1}{1 + 10^{(r2-r1)/400}}\right) \qquad (2)$$

where w is 1 for a win, 0.5 for a draw, and 0 for a lose. K is typically 32, although for the USCF rating system it diminishes for higher-rated players. In the USCF rating system additional modifications are made for new players and to keep the rating of the average rated member to 1500 USCF.

While the standard rating method can be used for determining robot performance, an alternate, less time-consuming, method is to measure the time it takes the program to solve a series of known test cases. For example, Kuszmaul estimates Startech's rating with Kaufman's "Rate your own computer" [33]. The best 20 times for solving 25 chess positions are summed together and a rating[5] is estimated as follows:

$$USCF\,Rating \;\approx\; 200 \times log_{10}\frac{1}{t} + 2930 \qquad (3)$$

where t is the summed time. Thus a factor of 10 in performance is estimated to be worth about 200 points.

5.2 Performance Model

In previous sections of this paper I have presented performance in terms of USCF ratings and search speeds. In addition, I have described an array of knowledge enhancements. In order to qualitatively assess the evolution of chess robots from average players to grandmasters and beyond, I have developed a simple model of performance based on a search speed and a knowledge factor component.

[5] Kaufman only claims his test cases to be valid for the 2200-2600 USCF range.

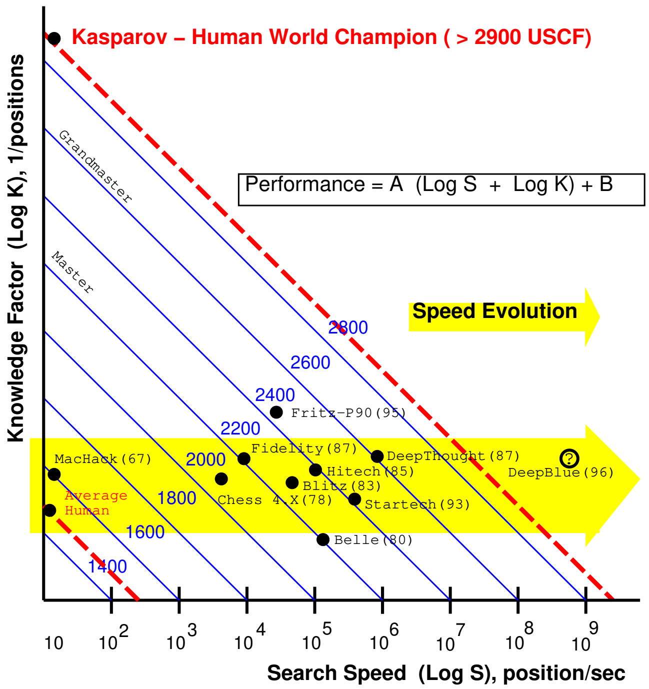

Figure 5: Chess Robot Performance

I start by decomposing the solution time in Kaufman's model as follows:

$$t \propto \frac{1}{S \times K} \qquad (4)$$

where S is evaluation speed in positions/sec and K is the knowledge factor. The units of K are 1/positions and it is a single number containing all non-speed-related performance components. K is proportional to:

- Accuracy of position evaluation
- Efficiency of move ordering
- Effectiveness of forward pruning and selective search
- Size of transposition table
- $\frac{1}{search\ overhead}$ in parallel searchers
- opening book and endgame database
- time control strategy

Substituting for t in Equation 3, we get an equation of the following form:

$$Performance = A(LogS + LogK) + B \qquad (5)$$

where $A = 200$ and $B = 2930$ for Kaufman's experiments. For my model I keep the $A = 200$ assumption, which is supported by similar work by Thompson [18], but I leave B as an unknown constant.

5.3 Comparison

Using Equation 5 with $A = 200$, I plot the knowledge factor of the machines in this study as a function of their speed and performance rating. In Figure 5, the diagonal lines correspond to constant USCF ratings, ranging from average human players (1500 USCF) to human World Chess Champion (2900+ USCF). By taking Deep Thought's speed of 900,000 positions/second and a rating of about 2600 USCF, we can determine its position on the graph and its relative knowledge factor. I intentionally leave out the scale of the Y-axis as the absolute meaning of knowledge differences is unclear.

The information to glean from this synthesis is that chess robots have evolved to faster and faster brute force searchers, with practically no increase in overall knowledge content. Exceptions to this are state-of-the-art microcomputer programs, represented by the Fritz-P90 data point, which have achieved a comparatively high knowledge factor. On the other extreme Belle, with a low knowledge factor largely due to its inefficient move ordering, was probably an extension too far into the brute-force direction. In addition, I note that while the computer chess community considers Deep Thought's chess-specific knowledge to be lower than Hitech's, factors such as an effective selective search algorithm push its K factor above Hitech's. When comparing Startech to Hitech, we see that the overall knowledge factor has decreased significantly due to the parallel search overhead. This decrease may imply that a parallel implementation such as Startech should pursue a higher knowledge level rather than simply parallelize a sequential implementation.

The high level bit to note is the direction of evolution of chess robot performance in contrast to man's performance. While the machines are only making small leaps in terms of increasing their knowledge, they are capable of matching (and possible surpassing in Deep Blue's case) man's performance with speed alone. They do not have to be more intelligent if they can just go faster. However, note that while the overall 1000 point increase in rating from MacHack to Deep Thought has been largely due to speed, knowledge is still important. A performance rating difference of over 300 points, equivalent to a factor of 13 in speed, can be attributed to the knowledge difference between the dumbest (Belle) and the smartest (Fritz) machines.

5.4 Model Limitations

This model assumes a constant $A = 200$. If A is changed the relative knowledge factors of the robots will change, potentially changing even the rank. Additionally, A is not necessarily constant and indeed some research has shown A to be larger for lower rated machines [13]. My main goal with this simple model is to contrast the overall evolution of chess machine knowledge and speed with the knowledge and speed of human chess players.

6 A Reconfigurable Chess Machine

In this section I analyze the feasibility of constructing a chess search engine out of FPGAs. However, instead of treating the FPGAs as random hardware components, as they are used in the Deep Thought and Hitech designs, I propose to treat an array of FPGAs as a *virtual machine* (Figure 6). The special-purpose hardware components of the search engine can potentially be replaced with a general-purpose virtual machine with only a small sacrifice in search speed. In addition, any speed lost may be more than compensated by taking advantage of the reconfigurable nature of this new system.

6.1 Computing with FPGAs

Today FPGAs are typically used as glue-logic in hardware designs. However, as FPGA sizes have increased and software has become available to take advantage of large arrays of low-cost FPGAs, it is now possible to construct an entire machine out of FPGAs. With the use of Virtual Wires [7] technology, an array of FPGAs can be programmed as a single, large FPGA. If such an array is programmed with the use of an algorithmic hardware description languages such as Behavioral Verilog, it is effectively a custom computing machine. In the remainder of this section I propose research directions for using this technology to construct a chess robot.

6.2 Reconfigurable versus Dedicated Logic

A first research step is to determine the overhead of using FPGA logic versus a custom logic design. In terms of silicon area the FPGA will both be slower and larger. Because the move generator and incremental evaluation circuits are not particularly large and have known functionality, a Verilog implementation can be designed and mapped to FPGAs. Further research should attempt to answer the following questions:

- What is the possible search speed with reconfigurable logic? Speed can be compared to hardware only implementations, software-only implementations, and re-

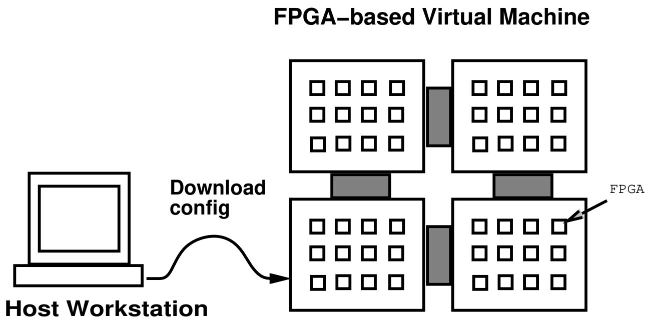

Figure 6: FPGA Computing

configurable implementations with a hardware move generator.

- What is the effect on slow evaluation complexity of reconfigurable logic? Should the slow evaluator be more complex or less?
- What is the cost/performance ratio in comparison to software only implementations? Since both FPGAs and processors are commodity parts, this number will be meaningful.

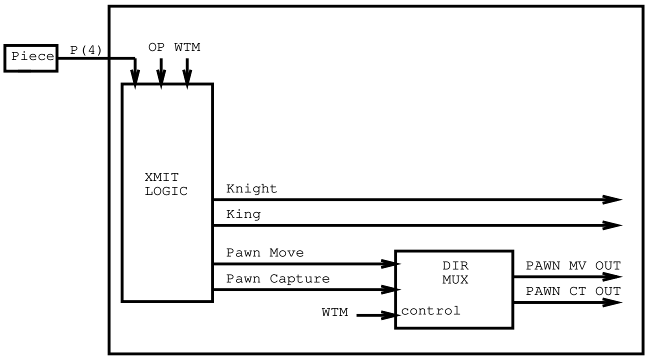

Figure 7: Specialized Move Generation Transmitter

6.3 Taking Advantage of Reconfigurability

Because each search proceeds for a substantial period, the FPGA logic can be completely re-programmed at the start of each search if desired. While there is not time to compile new FPGA logic between searches, a pre-determined number of FPGA configurations can be kept in storage and downloaded as needed. Hitech and Deep Thought use dynamic evaluation weights, and this concept can be easily extended to a virtual machine based design. Not surprisingly, the move generator is also a candidate for dynamic configuration. I consider this possibility in the next section.

6.3.1 Specialized Move Generator

In chess, certain events, such as pushing a pawn or capturing a piece, provide information about the remaining game positions. If the Black Queen is captured, then there will be no more moves by that Queen [6]. This information can be used to specialize the move generator. Examining the transmitter circuit in Figure 3, we see that the manhattan, diagonal, knight, pawn move, and pawn capture outputs may each be unnecessary with certain combinations of pieces. In addition, if we are only searching a fixed depth, for example 4 levels, then the kings can not reach all squares. Thus a set of specialized move generators can be created, with the appropriate generator loaded when needed. For example, Figure 7 shows a new transmitter circuit for positions with no queens, rooks, or bishops remaining. Since the circuit is combinational, the specialization can be implemented by fixing the appropriate input conditions and applying standard logic optimization. These specialized circuits will be smaller and faster than their generic counterparts.

7 Conclusion

In a famous forecast by Herbert Simon in 1957, "within 10 years a digital computer would routinely beat the world's best players". Although his prediction was not timely, it does appear that computers will eventually dominate human chess-playing ability. The remarkable feat is that chess robots can potentially win with increases in brute force alone! In this paper I have followed the evolution of robots from early conceptions to grandmaster level machines. I have developed a simple performance model that highlights this speed-dominated evolutionary path. In analyzing the anatomy of the robots Hitech, Deep Thought, and Startech, I have determined that fine-grain, hardware-level parallelism and specialization as well as coarse-grain, alpha-beta search level parallelism is benefitial for increasing robot speeds. For future research I proposed building a reconfigurable chess machine out of commodity FPGAs and taking advantage of dynamic reconfiguration. May the fastest machine win!

[6] I ignore the exception of a black pawn promoting - we can always re-load the initial move-generator.

References

[1] B. Abamson. An Explanation of and Cure for Minimax Pathology. Technical Report CSD-850034, UCLA, 1985.

[2] G. Adelson-Velskiy, V. Arlazarov, and M. Donskoy. Some Methods of Controlling the Tree Search in Chess Programs. *Artificial Intelligence*, 6:361–371, 1975.

[3] S. Akl, D. Barnard, and R. Dorna. Simulation and Analysis in Deriving Time and Storage Requirements for a Parallel Alpha-Beta Algorithm. In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 231–234, 1980.

[4] I. Althofer. Data Compression Using an Intelligence Generator: The Storage of Chess Games as an Example. *Artificial Intelligence*, 52:109–113, 1991.

[5] T. Anantharaman. *A Statistical Study of Selective Min-Max Search in Computer Chess*. PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, February 1990.

[6] O. Babaoglu. A Hardware Move Generator for Chess. Technical report, University of California, Berkeley, 1977.

[7] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 142–151, Napa, CA, April 1993. IEEE. Also as MIT/LCS TM-491, January 1993.

[8] G. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. PhD thesis, Carnegie Mellon University, Computer Science Department, 1978.

[9] D. Beal. Experiments with Tte Null Move. In *Advances in Computer Chess 5*, pages 65–79, Amsterdam, 1989. Elsevier Science Publishers. D.F. Beal, editors.

[10] H. Berliner. A Chronology of Computer Chess and Its Literature. *Artificial Intelligence*, 10:201–214, 1978.

[11] H. Berliner and C. Ebeling. Pattern Knowledge and Search: The SUPREM Architecture. *Artificial Intelligence*, 38:161–198, 1989.

[12] H. Berliner and C. Ebeling. HiTech. In *Computers, Chess and Cognition*, pages 79–109, New York, 1990. Springer-Verlag. T.A.Marsland and J.Schaeffer, editors.

[13] H. Berliner, G. Goetsch, and M. Campell. Measuring the Performance Potential of Chess Programs. In *Advances in Computer Chess 5*, pages 13–29, Amsterdam, 1989. Elsevier Science Publishers. D.F. Beal, editors.

[14] H. Berliner and C. McConnell. B* Probability Based Search. Technical Report CMU-CS-94-168, Carnegie-Mellon University, 1994.

[15] P. Ciancarini. Distributed Searches: A Basis for Comparison. *Journal of the International Computer Chess Association*, December 1994.

[16] *The Connection Machine System: Programming the NI.* Thinking Machines Corporation, March 1992. Version 7.1.

[17] L. Coles. Computer Chess: The Drosophia of AI. *AI Expert*, pages 25–31, April 1994.

[18] J. Condon and K. Thompson. Belle Chess. In *Chess Skill in Man and Machine*, pages 201–209, New York, 1983. Springer-Verlag. P.W. Frey, editor.

[19] J. Condon and K. Thompson. Belle Chess Hardware. In *Advances in Computer Chess 3*, pages 45–54, Oxford, 1983. Pergamon Press. M.R.B.Clarke, editor.

[20] D. G. A. D. *Thought and Choice in Chess*. Mouton, The Hague, 1965. English Translation with Additions of the Dutch edition (1946).

[21] D. Dalkenhol. Data Compression in Encoding Chess Positions. *Journal of the International Computer Chess Association*, September 1994.

[22] C. Ebeling. *All The Right Moves: A VLSI Architecture for Chess*. PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, 1986.

[23] A. E. Elo. *The Rating of Chess Players, Past and Present*. Arco Publishing, New York, 1978.

[24] R. Feldmann, P. Mysliwietz, and B. Monien. A Fully Distributed Chess Program. In *Advances in Computer Chess 6*, pages 1–27, Oxford, 1990. Pergamon Press. D.F.Beal, editor.

[25] R. Finkel and J. Fishburn. Parallelism in Alpha-Beta Search. *Artificial Intelligence*, 19:89–106, 1982.

[26] J. Gillogly. The Technology Chess Program. *Artificial Intelligence*, 3:145–163, 1972.

[27] G. Goetsch and M. Campbell. Experiments with the Null-Move Heuristic. In *Computers, Chess and Cognition*, pages 159–168, New York, 1990. Springer-Verlag. T. A. Marsland and J. Schaeffer, editors.

[28] R. Greenblatt, D. E. III, and S. Crocker. The Greenblatt Chess Program. In *Computer Chess Compendium*, pages 56–66, New York, 1988. Springer-Verlag. D.N. Levy, editors.

[29] F.-H. Hsu. A Two-Million Moves/s CMOS Single-Chip Chess Move Generator. *IEEE Journal of Solid-State Circuits*, SC-22(5), October 1987.

[30] F.-H. Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess*. PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, February 1990.

[31] R. M. Hyatt and H. L. Nelson. Chess and Supercomputers: Details About Optimizing Cray Blitz. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 354–363, May 1990.

[32] C. Joerge and B. Kuszmaul. Massively Parallel Chess. In *Third DIMACS Parallel Implementation Challenge*, Rutgers University, October 1994.

[33] L. Kaufman. Rate Your Own Computer - part II. *Computer Chess Reports*, 3(2):13–15, 1992-93.

[34] D. Knuth and R. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6:293–326, 1975.

[35] D. Kopec and I. Bratko. The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. In *Advances in Computer Chess 3*, pages 57–72, Oxford, 1982. Pergamon. M.R.B.Clarke, editor.

[36] A. Kotok. A chess playing program for the ibm 7090. MIT Bachelor's Thesis, 1962. also AI Project Memo 41, MIT Computation Center, 1962.

[37] B. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Massachusetts Institute of Technology, Department of Computer Science, May 1994.

[38] B. Kuszmaul. The StarTech Massively Parallel Chess Program. *Journal of the International Computer Chess Association*, March 1995.

[39] D. Levy and M. Newborn. *How Computers Play Chess*. Computer Science Press, New York, 1991.

[40] T. Marsland and M. Campbell. Parallel Search of Strongly Ordered Game Trees. *Computing Surveys*, 14(4), December 1982.

[41] J. McCarthy. Chess as the Drosophila of AI. In *Computers, Chess and Cognition*, pages 227–237, New York, 1990. Springer-Verlag. T.A. Marsland and J. Schaeffer, editors.

[42] A. Newell, C. Shaw, and H. Simon. Chess Playing Programs and the Problem of Complexity. *IBM Journal of Research and Development*, 2:320–335, October 1958.

[43] J. Nievergelt. Information Content of Chess Positions: Implications for Game-Specific Knowledge of Chess Players. In *Machine Intelligence 12*, pages 283–289, Oxford, 1991. Pergamon Press. J.E. Hayes, D. Michie, E. Tyugu, editors.

[44] A. Palay. The B* Tree Search Algorithm – New Results. *Artificial Intelligence*, 19(2), 1982.

[45] J. Pearl. Asymptotic Properties of Minimax Trees and gGame-Searching Procedures. *Artificial Intelligence*, 14:113–138, September 1980.

[46] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, Massachusetts, 1985.

[47] J. Schaeffer. The History Heuristic. *Journal of the International Computer Chess Association*, 6(3):16–19, 1983.

[48] C. Shannon. Programming a Computer to Play Chess. *Philosophical Magazine*, 41:256–275, 1950.

[49] D. Slate and L. Atkins. Chess4.5: The Northwestern University Chess Program. In *Chess Skill in Man and Machine*, pages 82–118, New York, 1977. Springer-Verlag. P.W. Frey, editor.

[50] F. Su, T. Anantharaman, M. Campbell, and A. Nowotzyk. Deep Thought. In *Computers, Chess and Cognition*, pages 55–78, New York, 1990. Springer-Verlag. T.A.Marsland and J.Schaeffer, editors.

[51] A. Turing. *Chess. Digital Computers Applied to Games*. 1953. Bowden, editor.

[52] H. Vigneron. Robots. In *Chess and Computers*, pages 14–23, 1990. D. Levy, editor, (translation of "Les Automates", La Nature, 1914, pp 56-61).

[53] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1984.

[54] A. Zobrist. A Hashing Method with Application for Game Playing. Technical Report 88, University of Wisconsin, 1970.