Multigame — A Very High Level Language for Describing Board Games *

John Romein Henri Bal Dick Grune

Vrije Universiteit
Department of Mathematics and Computer Science
Amsterdam, The Netherlands

{*john,bal,dick*}*@cs.vu.nl*

Abstract

Languages with implicit parallelism are easier to program in than those with explicit parallelism, but finding and efficiently exploiting parallelism in general-purpose programming languages by parallelizing compilers is hard. A compiler for a Very High Level Language, designed for a specific application domain, has more knowledge about its application domain and may use this knowledge to generate efficient parallel code without requiring the programmer to deal with explicit parallelism. To investigate this idea, we designed *Multigame*, a Very High Level Language for describing board games. A *Multigame* program is a formal description of the rules of a game, from which the *Multigame* compiler automatically generates a parallel game playing program.

Keywords: implicit parallelism, Very High Level Language, game-tree searching

1 Introduction

Parallelism in general-purpose languages can be either explicit or implicit. Explicit parallelism requires the programmer to explicitly point out in a program where the parallelism is, and deal with issues like communication, synchronization, deadlock prevention and data- and workload distribution. This is hard to do correctly and places a burden upon the programmer.

With implicit parallelism, the compiler tries to find the parallelism in a sequential program. Sequential programs are much easier to write, but finding and efficiently exploiting parallelism in general-purpose languages is beyond the state-of-the-art in compiler technology.

Yet we want to combine the advantages of easy programming and efficient execution. Compilers for Very High Level Languages, designed for some specific application domain, may be able to achieve this by exploiting their specific knowledge about the application domain. The price we pay is that the language is not general purpose. A well known example of such a language is *make*, for which several parallelizing interpreters exist (*GNU-make*, *Pmake* [21]).

Our research is directed to such Very High Level Languages. In this paper, we introduce one such language, *Multigame*, as an initial case study. *Multigame* is a language for describing board games. A *Multigame* program consists of a formal description of the rules of a game.

* This research is supported by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

The *Multigame* compiler analyzes these rules and generates a (parallel) program that plays the game. A *Multigame* program is completely free from any statement that expresses parallelism, data distribution, communication, or synchronization; the compiler knows how to generate code that exploits parallelism.

The organization of this paper is as follows. Section 2 explains the basics of the *Multigame* language. In Section 3 we elaborate on game-tree searching, and in Section 4 parallel and distributed search are described. In Section 5 we discuss the current state of the project and future work and draw some conclusions.

2 The Multigame language

Multigame can handle a restricted class of games. All games must be played on a board using pieces, thus card games like *bridge* and computer games like *tetris* cannot be expressed. The game must be played by either one player (e.g., *15-puzzle*) or two players (*chess*, *checkers*, *connect-4*, *othello*). All players must have perfect information, so *stratego* is excluded because the players cannot see the identity of the opponent's pieces. There are no dice, so backgammon can not be expressed either.

A *Multigame* program is a description of the rules of a game. A program consists of declarations (e.g., the dimensions of the board and the names of the pieces), and a description of the valid moves.

```
dimensions (3,3)

pieces
{
    mark            'X' 'O'
}

main =              try new_mark else draw.

new_mark =          find empty field,
                    replace by own mark,
                    try [ test three_in_a_row, win ].

three_in_a_row =    find own mark,
                    alldir,
                    repeat 2 times [ step, points at own mark ].
```

Figure 1: *Multigame* program for tic-tac-toe.

Figure 1 shows a complete *Multigame* program describing the rules of *tic-tac-toe*. The first line specifies that the game is played on a 3 by 3 square board. The pieces declaration declares one piece, named mark, represented as an X for the white player and an O for the black player.

The rest of the program defines the valid moves, using a combination of a Logo- and Prolog-like paradigm. There are some internal variables: a cursor called the *finger*, a *current direction*, and a *hand* to hold a piece. The finger points at one of the fields and the current direction can be set in any of the directions north, north-west, west and so on. A step command moves the finger one field in the current direction. The field that is currently pointed at can be tested for a particular piece, replaced by another piece, or the piece can be picked up and held in the hand, and (after moving the finger to another square) be put down.

Although the semantics of the rules are Prolog-like (in fact, the compiler generates full backtracking code) it is perhaps easier to think about them in terms of sets. Each statement accepts a set of positions and applies a test or modification to each position in the input set. For each position for which the operation succeeds, the resulting positions are placed in the output set. Consecutive statements act as function composition; the function main accepts a singleton input position set and computes the set of positions that can be reached by doing a valid move.

In the example above, the try statement in main tries to place a new mark onto the board. If new_mark fails (i.e. produces an empty set) the game is a draw. The rule new_mark is defined as follows. The statement find empty field produces a set of positions, with the finger associated with each position initialized to point at a different empty field. Then the field that is currently pointed at is replaced by a mark of our own color, i.e. the color of the player who is to make a move. Then we check whether we won the game. If the test three_in_a_row succeeds (i.e. is non-empty), we mark the board to be a winning leaf node. Otherwise, the move is still valid, but not a win. The rule three_in_a_row succeeds if it can find a piece of our own color, such that in some direction two consecutive steps can be done while still pointing at a piece of our own color. The step statement fails if we step off the edge of the board, and the points at own mark statement fails if we point at an empty field or a field occupied by the opponent.

As another example, consider the rules for a knight and bishop move in *chess*, shown in Figure 2. We will use the chess position of Figure 3 to illustrate the *Multigame* rules.

```
knight_move =  find own knight,
               pickup,
               orthogonal,
               step,
               either rotate 45 or rotate -45,
               step,
               not points at own piece,
               putdown.

bishop_move =  find own bishop,
               pickup,
               diagonal,
               repeat 0 .. infinity times [ step, points at empty field ],
               step,
               not points at own piece,
               putdown.
```

Figure 2: Example *chess* moves

Assuming that the pieces have been declared properly, the find statement tries to find a knight of our own color. If white is to make a move, it will fail, because there are no white knights. However, if black is to make a move, find succeeds in one way, with the finger initialized pointing at b8.

pickup removes the knight from the board and puts it in the hand. orthogonal continues with four different directions: north, east, south and west. The step in northern direction fails, but the others succeed. The fact that the finger of one of the positions points at the white pawn is no problem. The either statement changes for each of the input boards the direction 45 degrees clockwise and counterclockwise, resulting in six different possibilities. Three of them fail at the

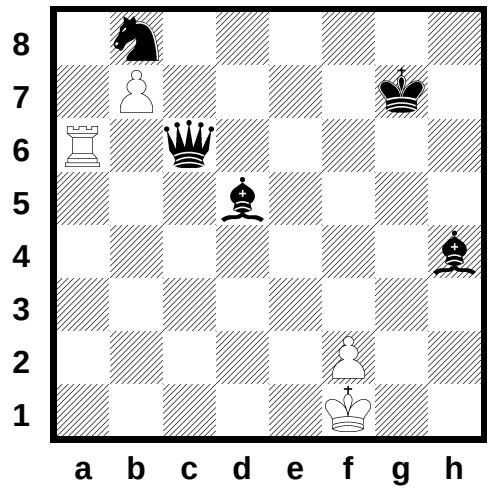

Figure 3: Example chess position

second step statement. The others have their fingers pointing at a6, c6 and d7 respectively. Then we test whether we are pointing at any piece of our own color, causing the board with the finger pointing at c6 to fail. Finally we put down the knight that we still held in our hand, effectively capturing the white rook at a6, or occupying the empty field d7.

Analogously we can define a bishop move. First we search for bishops. In the example above, we would find two black bishops, so we continue with two positions, with the fingers pointing at d5 and h4 respectively. In each case, we pick up the bishop, and set the direction to NW, NE, SW, and SE, yielding a total of eight different possibilities. The repeat statement states that we may make zero or more steps over empty fields. Despite the infinity, this statement terminates, because the finger will eventually step off the board (or point at a non-empty field). We must do one additional step, such that we do not capture a piece of our own color. Finally we put down the bishop that we have in our hand.

The rule definitions are allowed to be recursive. The language also features more exotic constructs to make it possible to correctly specify many "strange" rules like en-passant captures, but these are beyond the scope of this paper.

3 Game-tree searching

The *Multigame* compiler generates a game playing program from a formal rule description of a game (see Figure 4). The compiler consists of a *front-end compiler* and a *library*. The front-end compiler generates game-specific code, such as a description of a board and a move generator. The library provides functions that are common to several or all games. An optional game-dependent evaluation function can be provided by the programmer and linked with the rest of the program. The front-end compiler generates ANSI-C code, which is portable and efficient. The most important components of a game-playing program (the *move generator*, *search algorithms*, *heuristics*, *platform support*, and *evaluation functions*) are described in the remainder of this paper.

Game playing programs search a game tree by repeatedly thinking moves ahead (see Figure 5). For one-player games (e.g., the *15-puzzle*) the shortest path from a problem instance to a solution is searched. For two-player games, the move leading to the best obtainable position is searched, assuming that the opponent replies with the best countermove. Except for some trivial games,

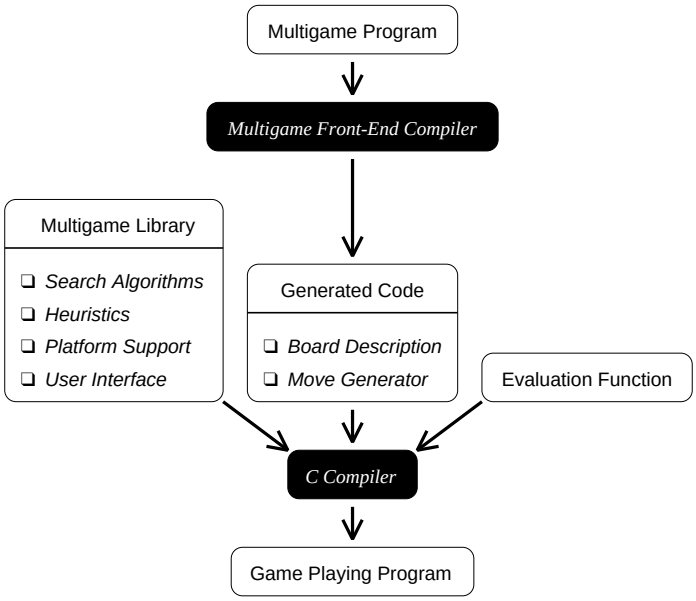

Figure 4: Structure of a game-playing program.

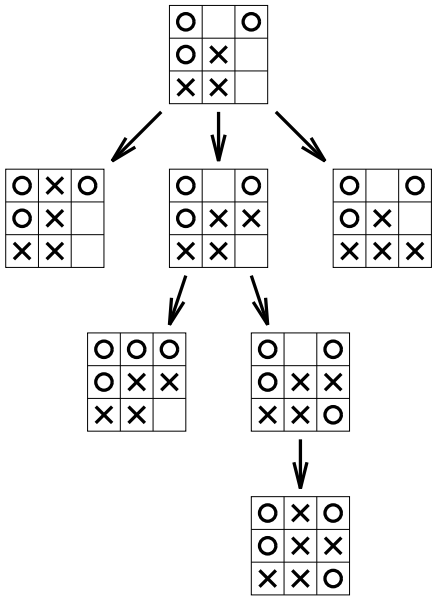

Figure 5: An example game tree.

the game tree is too large to be fully searched. Therefore, the search algorithm uses a number of game-dependent and -independent heuristics to guide the search, and restricts the number of moves thought ahead.

The move generator accepts a position and computes the set of all positions that can be reached by doing a valid move. This depends very much on the rules of the game. Therefore the move generator itself is in full generated by the front-end compiler from the rules of the game. For each statement, the compiler generates code to execute the statement and backtracking code to undo the effects of the operation. This reflects the similarity with the Prolog paradigm, although Prolog programs usually are interpreted and *Multigame* programs are compiled.

Each node in a game tree is either a *leaf node*, a *frontier node*, or an *internal node*. Leaf nodes distinguish themselves from frontier nodes in that they represent a won, drawn and lost position in the tree; frontier nodes are leafs in the tree but not final positions in the game.

The search engine repeatedly selects a frontier node for expansion by the move generator. The order in which the nodes are expanded depends on the search algorithm. Examples of tree search algorithms are α-β search [11] and Proof-Number search [1] for two-player games, and IDA* [12] for one-player games. Except for the fact that a search strategy is either for one- or for two-player games, search strategies are independent of the game being played, so they can be found in the *Multigame* library. Other search algorithms can be added easily.

Some search strategies require a static board evaluation function in order to play well. For example, α-β uses an evaluation function that indicates whether or not the position is advantageous for the player who is to make a move. IDA* works best if the evaluation function gives an accurate lower bound of the distance to the final goal. Proof-Number search does not use an evaluation function at all (although an evaluation function can be used to estimate proof- and disproof-numbers). The evaluation function therefore is not only game-specific, but also search-strategy dependent.

Ideally, we would like the front-end compiler to generate appropriate evaluation functions from the rules of the game, but this is very hard to do [16] and is not part of our research. Therefore, the programmer can optionally provide a hand-written evaluation function for a specific game to

improve the quality of the playing program. We have ported evaluation functions for checkers and othello. The checkers evaluation function was ported from Chinook, the current man-machine world champion [18]. The othello evaluation function was ported from Aïda (a program developed by Jacco Gnodde and Dennis Breuker at the University of Leiden). For the 15-puzzle we use the Manhattan distance function [12].

The search engine can use a variety of game-independent heuristics to guide the search. Examples of important and well known heuristics are *iterative deepening* [19], *transposition tables* [9], and the *history heuristic* [17]. These heuristics are mostly game independent, so they can be found in the library. The library can easily be extended with other heuristics.

In the transposition (hash) table, information is stored about previous searches of (sub)trees. In order to look up a transposition table entry, for each position a 64-bit signature is maintained. Two identical board positions always yield the same signature; two different positions almost always yield different signatures. The front-end compiler generates efficient code to compute a signature [22], by generating a table with 64-bit random numbers, indexed by *field-number*, *piece-number*, and *color* (for two-player games). Each time a piece is placed upon or removed from the board, the board's signature is XORed by the piece's corresponding table entry. We use a two-level replacement algorithm when a transposition entry conflict occurs [6].

The best move from one position is often the best move from a similar position. The history heuristic [17] records how frequently a possible move is the best move, and sorts children of interior nodes in descending order of frequency. The most promising move is searched first. Normally the heuristic is implemented using two matrices (one for each player), indexed by *from-fieldnumber* and *to-fieldnumber*. For a general *Multigame* program this is not possible, because valid moves are not restricted to moving exactly one piece from one field to another. We solved this problem by maintaining a list of (*signature-difference*, *value*) pairs, where the signature-difference is the signature of the position *from* which a move is made XORed with the signature of the position *to* which a move is made. Due to the way signatures are computed [22], a knight move from d5 to c7 always yields the same signature-difference, irrespective of the pieces on the other fields. Access to the list is sped up by hashing.

4 Parallel and distributed game-tree search

Usually a game playing program can improve its quality by searching the game tree deeper. Because of the exponential growth of the tree, searching one ply deeper takes an order of magnitude more time. A solution is to let multiple processors cooperate to reduce the total search time.

One of the most important aspects in our research is to investigate how we can exploit parallelism in *Multigame* programs in a transparent way. We assume that *Multigame* programs will be run on distributed-memory multicomputers (e.g., a collection of workstations), so we try to exploit coarse-grained parallelism. It is also possible to generate code for shared-memory systems.

Our department has a distributed system at its disposal, consisting of 80 SPARC Classic clones, each equipped with 32 Mb RAM. This amounts to a total of 2.5 Gb memory. Each board has an Ethernet network controller, but is diskless and not connected to a terminal. A few additional boards provide disk services. The nodes are connected through a 10 Mbit/sec segmented Ethernet. The system runs the Amoeba Distributed Operating System [20]. We use this system to run our experiments.

Parallelism can be found in several components of the game playing program. However, most parallelism is too fine-grained. Only the search engine provides a reasonable amount of coarse-grained parallelism, which can be exploited by having multiple processors searching independent

subtrees. The sequential search algorithms have to be modified to do parallel search. This is easy for IDA*, but it is not obvious how to parallelize α-β [8] and Proof-Number search efficiently.

Multicomputers do not only provide more processing power, but also more memory. This memory can be used for keeping larger game trees in main memory, which is important for best first search algorithms. Each machine searches some subtree and keeps this subtree in local memory. The memory can also be dedicated to a larger transposition table, thus lowering the number of conflicts and decreasing the number of unnecessary tree searches. The table is partitioned, so each machine has part of the table in its local memory. This means that a lookup or store usually requires communication.

Parallel game-tree search introduces three kinds of overheads [2]: *search overhead*, *synchronization overhead*, and *communication overhead*. Search overhead is caused by the parallel search algorithm searching more nodes than a sequential algorithm would. Synchronization overhead occurs when a node must wait until the computations for its children have been finished. Communication overhead results from the latency of messages that are sent from one processor to another.

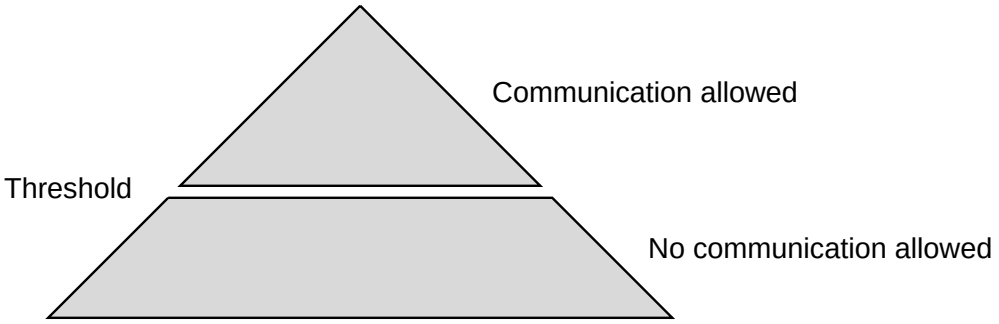

Figure 6: Communication only allowed in the upper part of the tree.

Usually, trying to lower one kind of overhead increases the others. For example, one can omit a remote transposition table lookup to save a few messages, but this can cause a larger part of the tree being (re)searched. We use a scheme where we only do communication if there is a potentially large gain by doing so. Forking off work is only done for nodes near the root of the tree (see Figure 6), because splitting off work too far from the root would not keep the remote processor busy for a long time. Similarly, remote transposition table lookups are done only for nodes below a certain tree level; then a hit may save a lot of work. Above this threshold, a potential hit may save less time than the actual communication costs, so we omit the remote transposition table lookup and just search the small subtree.

Whenever a processor becomes idle, it asks a random processor for work. This processor either answers that it has no work, or selects a frontier node in its own subtree so that it can be searched by the idle processor. Special care must be taken at global synchronization points, when there is no work for processors other than the processor that owns the root of the tree. It must be avoided that the network is flooded by request-for-work messages. A processor should wait a short time after several consecutive requests for work failed.

There are many ways to parallelize α-β [14, 3, 5]. We use the Youngest Brothers Wait First algorithm, described in [7]. The goal of this algorithm is to reduce the search overhead. The children of a node may be evaluated in parallel if and only if its first successor has been evaluated completely. This forces a reasonable bound for α, so that the forked-off children are not searched with an unnecessarily wide α-β window.

On our system, a thread is blocked for tens of thousands of cycles during interprocessor

communication. Meanwhile, we want to keep the processor busy searching other parts of the tree. Therefore, it is possible to have multiple threads per processor searching different parts of the tree. This introduces an additional amount of parallel search- (and thread switching) overhead, so the number of concurrent threads per processor should be limited to a few.

The *Multigame* library contains some modules with platform-specific code for Unix (BSD and System V), Solaris, Pthreads [15], Amoeba [20], and Panda [4]. Three types of search are supported: *sequential*, *multi-threaded*, and *distributed* search. Multi-threaded search algorithms use a platform-independent interface for threads, mutexes, semaphores and condition variables. The Solaris and Pthreads modules provide a bridge between this platform-independent interface and the Solaris resp. Pthreads interfaces. In addition to threads and synchronization primitives, the Amoeba and Panda modules analogously provide a platform-independent interface for Remote Procedure Calls and Group Communication [10] for distributed search. The standard Unix platform has no provision for parallel search. These platform-independent interfaces simplify adding new platforms. Besides, Panda itself provides a platform-independent interface, so a *Multigame* program will run on any platform that is supported by Panda.

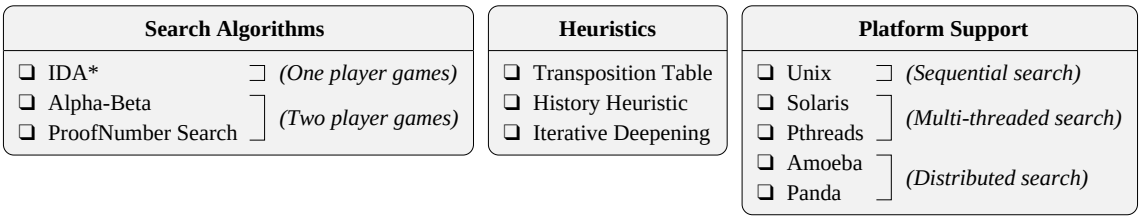

Figure 7: Selectable library components.

Figure 7 summarizes the selectable library components: Search algorithms, heuristics, and platform support. When generating a game-playing program, one specifies as compile-time options a search algorithm, a set of usable heuristics, and a platform on which the program should run. Not all combinations are viable, for example Iterative Deepening is always required for IDA*.

An important problem is that different games require different adjustments in order to play well. For example, the combination of a transposition table and a history table is very useful for playing *chess* using α-β search, but the history heuristic is completely useless for playing *othello*. On distributed systems the optimal adjustment may even depend on the network topology that is used. A topic for future research is to see whether an optimal adjustment can be found automatically, for example by using genetic algorithms.

5 Current state, future work, and conclusions

We have developed *Multigame*, a Very High Level Language for describing board games. A *Multigame* program describes the rules of a game, and does not have any explicit commands to specify parallelism, work distribution, synchronization, communication, or deadlock prevention. The *Multigame* compiler has enough knowledge about its application domain to generate a parallel program, so the programmer need not think about parallelism, which simplifies the task of programming. Games that can be described in *Multigame* include: *chess*, *checkers*, *othello*, *connect-4*, *nine-man's-morris*, and the *15-puzzle*.

The current state of the project is as follows. A *Multigame* front-end compiler has been implemented which accepts the entire language. Several heuristics (history heuristic, transposition table) and sequential search algorithms (α-β, IDA*, Proof-Number search) have been implemented. We

have a prototype implementation of multi-threaded α-β and IDA* search, and we are currently writing code to do distributed search.

There are some topics for future research. Genetic algorithms might help to find optimal adjustments of game playing programs. Opening books and endgame databases may significantly improve the quality of a game playing program. Such databases usually are stored on disk, because they are very large. On parallel systems a problem arises if many processors simultaneously access the database: the response time can become as high as several seconds. To decrease the access time, we consider storing the database in the (distributed) main memory of our parallel system. Other processors can access the data by using RPCs. Our department possesses a distributed system that can easily hold a 1 Gb endgame database, and still leaves some storage for transposition tables. Another challenge is the automatic creation of such databases from the rules of a game. It requires a lot of computing time to generate these databases [13], which makes them suitable problems to be solved on a distributed system. Another topic for future research is to develop other Very High Level Languages. An example of a Very High Level Language for an application domain related to *Multigame* is a language for describing (parallel) search algorithms.

Acknowledgements

Raoul Bhoedjang, Dennis Breuker, and Tim Rühl provided useful comments on a draft version of this paper. Arnold Geels devised the Logo- and Prolog-like paradigms used in *Multigame*. We thank Jonathan Schaeffer for providing us with the sources of Chinook and Jacco Gnodde and Dennis Breuker for providing us with the sources of Aïda.

References

[1] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Rijksuniversiteit Limburg, Maastricht, the Netherlands, September 1994.

[2] E. Altmann, T.A. Marsland, and T. Breitkreutz. Accounting for parallel tree search overheads. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume III, Algorithms and Applications, pages 198–201, University Park, Penn., August 1988. Penn State.

[3] H.E. Bal and R. van Renesse. A summary of parallel alpha-beta search results. *ICCA Journal*, 9(3):146–149, 1986.

[4] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A portable platform to support parallel programming languages. *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, September 1993.

[5] P. Ciancarini. Distributed searches: A basis for comparison. *ICCA Journal*, 17(4):194–206, 1994.

[6] C. Ebeling. *All the Right Moves: A VLSI Architecture for Chess*. PhD thesis, Carnegie-Mellon University, Pittsburg, PA, 1986.

[7] R. Feldmann. *Game Tree Search on Massively Parallel Systems*. PhD thesis, University of Paderborn, August 1993.

[8] R.A. Finkel and J. Fishburn. Parallelism in alpha-beta search. *Artificial Intelligence*, 19:89–106, 1982.

[9] R.D. Greenblat, D.E. Eastlake III, and S.D. Crocker. The Greenblat chess program. In *Proceedings of the Fall Joint Computing Conference*, pages 801–810, San Fransisco, 1967.

[10] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, Amsterdam, December 1992.

[11] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[12] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[13] Robert Lake, Jonathan Schaeffer, and Paul Lu. Solving large retrograde analysis problems using a network of workstations. In H.J. van den Herik, I.S Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess VII*, pages 135–162. University of Limburg, Maastricht, Netherlands, 1994.

[14] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, December 1982.

[15] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, San Diego, CA, Winter 1993. USENIX.

[16] B.D. Pell. *Strategy Generation and Evaluation for Meta-Game Playing*. PhD thesis, University of Cambridge, August 1993.

[17] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[18] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53:273–289, 1992.

[19] D.J. Slate and L.R. Atkin. CHESS 4.5 — the northwestern university chess program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer Verlag, 1977.

[20] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J . Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[21] B.B. Welch. Measuring performance of caching in the Sprite network file system. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II)*, pages 229–247. USENIX, Atlanta GA, March 1991.

[22] A.L. Zobrist. A new hashing method with application for game playing. Technical Report 88, Computer Science Department, University of Wisconsin, Madison, 1970. Reprinted in: *ICCA Journal*, 13(2):69–73, 1990.