Multigame — An Environment for Distributed Game-Tree Search

VRIJE UNIVERSITEIT

Multigame — An Environment for Distributed Game-Tree Search

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen /
Wiskunde en Informatica
op 18 januari 2001 om 13.45 uur
in het hoofdgebouw van de universiteit,
De Boelelaan 1105

door

Johannes Willem Romein

geboren te Culemborg

Promotor: prof.dr.ir. H.E. Bal

Contents

List of Figures

List of Tables

Acknowledgments

The computer systems group at the Vrije Universiteit is an excellent place for doing research. The competent and friendly people in the group and an almost unlimited access to a large distributed system provide a pleasant working environment. I never could have done the work for this thesis on my own, and much of what we achieved is the result of discussions and cooperation with other people.

Despite the pleasant working environment, I never considered writing a thesis an enjoyable occupation. Before I started writing this thesis, I thought that spending a week and a half to discover the cause of a core dump was the worst that could happen during a four-year's research period. That was true, but after the four years were over, the majority of the thesis still had to be written (and according to the statistics, I need not be ashamed of that). Documenting the ideas of the past four years is not something to look forward to; it is even less attractive than hunting bugs. I was still full of ideas about how to improve the algorithms and make the software faster, and that makes it hard to stop implementing and start writing. Moreover, there seems to be so little progress while writing. Realizing that I finished 0.3% of the book after working an entire day did not make me happy at all — except for the last 0.3% ...

There are many people whom I would like to thank. In the first place, I want to thank Henri Bal. I cannot imagine a friendlier and better supervisor. He gave me the freedom to direct the research in the areas that I found interesting. He taught me the difference between doing research (which results in publications) and hacking (which is fun but useless). He taught me how to write. He reads and corrects texts that I wrote in a matter of days. I could always walk into his office whenever I felt an urgent need to see him, though he must be a busy person.

Dick Grune was a second supervisor. He seems to be knowledgeable in whatever computer-science-related area one can think of, and surely knows next to everything about programming languages and compiling techniques. He was closely involved with the design of the Multigame language.

It was pleasant to have Aske Plaat as roommate for about two years. We had countless discussions about game trees, airplanes, economics, more game trees, more airplanes, and more economics. Although I owe my current research fund to his early departure, I think it is a pity that he left. His constructive comments on a preliminary version of this thesis improved the quality and readability substantially.

I have great respect for Jonathan Schaeffer, one of the world's most learned people in the area of artificial intelligent game playing. The idea of Transposition Driven Scheduling (see Chapter 5) arose during a joint dinner in an Indonesian restaurant. He also gave numerous suggestions to improve both the contents and the presentation of this thesis.

Arie de Bruin and Jaap van den Herik form together with Dick Grune, Aske Plaat, and Jonathan Schaeffer the reading committee of this thesis. I would like to thank all members for their comments on the thesis.

I would also like to thank Raoul Bhoedjang. He wrote the first version of the Myrinet network processor software to support the fast remote transposition table accesses (described in Section 4.4.4) and spent weeks debugging the Myrinet software and hardware. Later he designed and implemented LFC, a general-purpose communication layer on top of Myrinet. Multigame uses the low latency and high (broadcast) bandwidth properties provided by LFC. He currently is at Cornell University; I hope he has found a new source of peanuts there.

Our group has three excellent programmers, Kees Verstoep, Ceriel Jacobs, and Rutger Hofman. Kees, with whom I kept our computer system in a healthy shape during the past few years, was closely involved in the development of LFC, and did most of the work porting the old network processor software to the current version that extends LFC. Rutger wrote the Panda library (together with Raoul Bhoedjang and Tim Rühl) and the *prun* scheduler for starting distributed applications on our computer system. Despite the complexity, these pieces of software grew to become mature and usable tools.

A number of other people directly or indirectly contributed to this thesis. A long time ago, Andy Tanenbaum suggested doing research on distributed game playing. Arnold Geels devised the (unpublished) GameSpeak language. Although the Multigame language is a completely new design, the idea to use the Logo programming paradigm was borrowed from GameSpeak. Geurt Jongbloed was helpful in explaining statistics. Thilo Kielmann, Koen Langendoen, and Tim Rühl are friendly (former) colleagues. Gregory Mounie, our new colleague from France, kindly read a close-to-final version of this thesis, and corrected the last few tyypos. The NWO (the Dutch organization for scientific research) financially supported part of the research.

Many people placed their software at our disposal. Jonathan Schaeffer provided us with the sources of Chinook, allowing us to use its evaluation function in our checkers implementation. Jacco Gnodde and Dennis Breuker provided us with the sources of Aïda, from which we ported an Othello evaluation function. Andrew Chien and Scott Pakin made the Illinois Fast Messages software available to us.

Finally, I want to thank my family. My parents raised me, gave me the opportunity to study, and heard my complaints about writing a thesis. My brother, Eric, spent countless hours in designing a logo for Multigame; the result is found on the cover.

And Elisabeth ? She is just the sweetest girl in the world (and surroundings).

Samenvatting

Dit proefschrift beschrijft Multigame, een omgeving voor het parallel (op meerdere computers tegelijk) zoeken van zogenaamde spel-bomen. Spel-bomen zijn bomen die ontstaan bij het vooruit kijken in bordspel-spelende programma's zoals schaken en Othello. Het dieper zoeken (meer zetten vooruit kijken) leidt i.h.a. tot beter spel, maar elke zet verder vooruit kijken kost, door de exponentiële groei van de boom, een factor meer tijd. Door meerdere computers tegelijk elk in een deel van de spel-boom te laten doorzoeken, kan tijd worden bespaard. Anders geredeneerd: wanneer een programma zo'n drie minuten heeft om een zet te doen, kan met een parallel systeem dieper worden gezocht dan op een enkele computer, zodat het programma beter speelt.

Tijdens het onderzoek dat aan dit proefschrift vooraf ging, hadden we twee doelstellingen voor ogen. In de eerste plaats wilden we het werk van de applicatie programmeur vereenvoudigen door een programmeer-omgeving aan te bieden waarbij de programmeur niet hoeft na te denken over parallellisme. Zaken als synchronisatie, communicatie en werk- en data-verdeling worden door het Multigame systeem zelf afgehandeld. In de tweede plaats wilden we onderzoekers in dit gebied een experimentele omgeving verschaffen om onderzoek te doen naar het parallel zoeken in spelbomen. Hierbij valt te denken aan onderzoek naar parallelle zoek-algoritmen en aan heuristieken en optimalisaties die het zoeken in een boom versnellen. Deze twee doelstellingen leidden tot het ontwerp en de implementatie van Multigame.

De Multigame omgeving bestaat uit een programmeertaal, een compiler, en een runtime-systeem. Het runtime systeem bevat spel-onafhankelijke software; de spelafhankelijke spelregels kunnen in de Multigame-taal worden uitgedrukt.

In hoofdstuk 3 behandelen we de Multigame-taal. Een Multigame-programma is een formele definitie van de regels van een bordspel. De programmeur beschrijft de regels van een spel in een Multigame-programma. De Multigame-compiler analyseert deze regels en creëert een zetten-generator aan de hand van de regels. Samen met het Multigame runtime-systeem vormt deze een programma dat het spel op een (parallelle) machine speelt. De programmeur kan de kwaliteit van het spel-spelende programma verbeteren door spel-afhankelijke heuristische informatie in de vorm van een evaluatie-functie (in C) toe te voegen.

De taal is gebaseerd op een combinatie van de Logo en Prolog programmeer-

paradigma's. Aan de hand van een cursor beschrijft de programmeur hoe stukken over een bord verplaatst kunnen worden. De programmeur hoeft zich niet bewust te zijn van het feit dat het programma op een parallelle machine draait.

We beargumenteren ontwerp-keuzes van de taal en beschouwen altenatieve taal-ontwerpen. We laten zien hoe de door de Multigame-compiler gegenereerde code werkt en analyseren de efficiëntie van de door de compiler gegenereerde code. We beschrijven onze ervaringen met de implementatie van de Multigame-compiler, die in een object-georienteerde taal (C++) is geschreven. We belichten hier ook de rol van evaluatie-functies.

Hoofdstuk 4 beschrijft het Multigame runtime-systeem. Het runtime-systeem is een veelomvattende laag software die spel-onafhankelijke functionaliteit bevat. We gaan in op de organisatie van het runtime-systeem en beschrijven de parallelle zoek-algoritmen die we hebben geimplementeerd. De werkdistributie is gebaseerd op het bekende werk-stelen: een processor die geen werk meer heeft, vraagt een willekeurige andere processor om werk en neemt een deel van diens werk over.

De transpositie-tabel speelt een prominente rol in dit hoofdstuk. Transposities zijn knopen in de zoek-boom met meerdere ouders (in feite is de term 'zoek-boom' misleidend, aangezien er grafen en geen bomen doorzocht worden). De transpositie-tabel is een cache die zoek-resultaten bevat van knopen die reeds doorzocht zijn. Een zoek-machine gebruikt de transpositie-tabel om te voorkomen dat delen van de boom meerdere keren doorzocht worden. We beschrijven drie soorten gedistribueerde implementaties: gepartitioneerde tabellen (iedere processor slaat een deel van de tabel op), gerepliceerde tabellen (iedere processor heeft een eigen kopie van de gehele tabel), en lokale tabellen (iedere processor heeft zijn eigen variant van de tabel, die niet consistent wordt gehouden met die op andere processoren).

Gepartitioneerde tabellen hebben als voordeel dat ze meer waarden kunnen bevatten naarmate er meer processoren worden gebruikt, maar als nadeel dat bijna elke lees- of schrijf-operatie communicatie met zich meebrengt. Bovendien zijn lees-operaties synchroon, d.w.z. dat een processor wacht tot het antwoord van een lees-verzoek binnen is. De hoge wachttijd is grotendeels te weten aan de trage manier waarop een lees-verzoek op de binnenkomende machine wordt afgehandeld. Het interrupt-mechanisme om de processor van een binnenkomend bericht op de hoogte te stellen is te tijdrovend. Het alternatief, de processor de netwerk-adapter regelmatig te laten vragen of er een bericht is binnen gekomen, heeft als nadeel dat binnenkomende berichten pas na verloop van tijd worden gezien en afgehandeld. Gedurende deze tijd zit de processor die het lees-verzoek verstuurde te wachten op antwoord. Om de wachttijd van een lees-operatie te verminderen hebben we gebruik gemaakt van de mogelijkheid om de firmware op de Myrinet netwerk-adapter aan te passen. We hebben de firmware zo veranderd dat de netwerk-processor het binnenkomend bericht zèlf afhandelt en de transpositie-tabel-waarde leest, in plaats van het bericht aan de CPU door te geven en het door de CPU af te laten handelen. We laten zien dat dit de wachttijd met 45–70% vermindert.

Om de wachttijd verder te verlagen, beschrijven we een experiment met het vroegtijdig en speculatief versturen van lees-verzoeken, zodat de antwoorden al klaarliggen of eerder terug zijn wanneer zij nodig zijn. Een lees-verzoek wordt verstuurd zodra het waarschijnlijk is dat het antwoord daadwerkelijk nodig is. De verzendende processor kan ondertussen verder gaan met rekenen tot het antwoord echt nodig is. Hoewel soms pas kort van te voren voorspeld kan worden welke tabel-waarden nodig zijn en vanwege het speculatieve karakter ook onnodige communicatie wordt verricht, kan de wachttijd op deze manier met nog eens 19–57% worden teruggebracht.

Gerepliceerde transpositie-tabellen hebben als voordeel dat elke lees-operatie lokaal is, maar schrijf-operaties moeten naar alle andere machines worden gecommuniceerd. Bovendien groeit het aantal waarden dat in de tabel kan worden opgeslagen niet wanneer meer processoren worden toegevoegd, zoals bij gepartitioneerde transpositie-tabellen het geval is.

Lokale transpositie-tabellen hebben als voordeel dat er geen tijd verloren gaat aan communicatie. Het nadeel is dat transposities die door verschillende processoren worden doorzocht niet als transposities herkend worden. Hierdoor kunnen gedeeltes van bomen meerdere keren worden doorzocht door verschillende machines, wat leidt tot extra rekenwerk.

De transpositie-tabel is niet de enige component die de prestaties van de zoekalgoritmen verbetert. We hebben ook andere (standaard) spel-onafhankelijke en spelafhankelijke heuristieken en optimalisaties geimplementeerd die het zoeken in de spelbomen versnellen en de prestaties van de applicaties verbeteren. Hieronder vallen bijvoorbeeld de “history heuristic” en databases die patronen in bordposities herkennen.

We geven een uitgebreide analyse van de prestaties van de verschillende componenten van de Multigame-omgeving en vergelijken de verschillende transpositie-tabel-implementaties voor zes verschillende applicaties. We laten zien dat niet één implementatie is die onder alle omstandigheden het beste werkt. Tenslotte beschrijven we welke problemen het implementeren van een parallel runtime systeem voor spel-spelende programma’s moeilijk maken.

Hoofdstuk 5 introduceert een nieuwe methode om spel-bomen voor 1-persoons spellen op zeer efficiente weze parallel te doorzoeken. Traditionele methoden, gebaseerd op het stelen van werk, zijn óf inefficiënt doordat ze transposities niet detecteren en daardoor sommige boom-gedeelten meerdere keren doorzoeken, óf inefficiënt doordat de extra communicatie voor het detecteren van transposities erg duur is. De nieuwe methode partitioneert de transpositie-tabel over de beschikbare processoren. Een nieuw gegenereerde knoop in de spel-boom wordt naar de processor gestuurd die de corresponderende transpositie-tabel-waarde kan bevatten, waar gecontroleerd wordt of het al dan niet om een transpositie gaat. Is dit niet het geval, dan wordt de knoop op de ontvangende processor geëxpandeerd, en worden de kinderen op hun beurt over de processoren verdeeld. Het grote voordeel van het nieuwe algoritme is dat alle communicatie asynchroon is (d.w.z. dat er nooit op antwoord hoeft te worden gewacht).

We hebben de prestaties van het nieuwe algoritme voor een aantal toepassingen vergeleken met verschillende varianten van de traditionele methoden. Het nieuwe algoritme werkt op 128 processoren tussen de 119 en de 124 keer zo snel als op een enkele processor, waar traditionele methoden tussen de 8.2 en 78 blijven steken.

Chapter 1

Introduction

Despite the many parallel programming languages that exist [6, 10, 109], parallel programming remains a difficult task. The alternative, writing a sequential program and letting the compiler find and exploit the available parallelism, is much easier for the programmer but this has so far only been successful for applications with regular parallelism. Yet we would like to offer the programmer a programming model that hides issues like communication, synchronization, work- and data distribution, and deadlock prevention.

We believe that this goal can be achieved with problem-solving environments that are designed for one particular application domain. Programmers can describe a problem in a dedicated, very high level language. The compiler and accompanying runtime system have knowledge about the application domain and can use this to exploit parallelism automatically. In these languages, generality is traded for an easier programming model and implicit parallelism.

To study the feasibility of such an approach we designed *Multigame*, a system for solving game-tree search problems. We chose this application domain, because it is an interesting and challenging area for parallel programming research, and because game-tree search, used to decide which move should be made from a given position, is an instance of the important class of heuristically informed search algorithms. Heuristic search is one of the cornerstones of Artificial Intelligence. Its applications range from logic programming to pattern recognition, from theorem proving to chess playing. For many applications, such as real-time search and anytime algorithms [39], achieving high performance is of great importance, both for solution quality and execution speed.

Playing a game well requires searching large game-trees in real-time; the underlying assumption is that deeper search improves the solution quality [76, 116]. Parallel and distributed architectures can contribute to better play; for example, Deep Blue [56, 57], which defeated Kasparov in 1997, is highly parallel. Lacking special-purpose hardware, grand-master chess knowledge, and the manpower to tune the program for years, our Multigame-based chess program would never be able to beat Deep Blue. However, Multigame can handle a broad class of single-agent and two-agent search problems, whereas Deep Blue handles only one application: chess. Like Deep Blue, Multigame is designed to provide *fast, parallel search on distributed (and shared) memory systems*.

A fair amount of research has been done on parallel game-tree search. Game-tree search is known to be hard to parallelize efficiently, rendering it a challenge to implement an efficient compiler and parallel runtime system. A prototype implementation is used throughout this thesis.

Multigame is designed with two research objectives in mind:

- provide the *application programmer* with an environment in which problems can easily be expressed and solved; and
- provide *researchers in this field* with an experimental environment to do research on distributed game-tree search.

The first research objective is to provide the application programmer with an environment for one and two-person board games that can be used to describe a game, *with minimal effort*, and *without compromising the playing strength of the program* (it is useless to parallelize an algorithm that plays weakly). Unfortunately, "minimal effort" and "without compromising the playing strength" hardly go together. "Minimal effort" implies that the system should require only a formal description of the rules of a game. "Without compromising the playing strength" means that from these rules, the system should determine a playing strategy that is as good as a program that carries the expert knowledge of a human master, something that is far beyond the current state-of-the-art in Artificial Intelligence (see also Section 2.3). Therefore, we allow the programmer to supply the system with human expert knowledge for a particular game, in the form of a game-specific evaluation function.

Nevertheless, Multigame greatly simplifies the programmer's job, since it offers a programming model that is free from explicit parallelism. The programmer formally specifies the rules of a game in the Multigame language and an evaluation function (in C). The Multigame system automatically generates a program that can play the game on a parallel system with shared or distributed memory. The language is designed in such a way that it is easy to describe the rules of board games, and its expressiveness allows the programmer to describe most existing board games. The Multigame system is usable for prototyping of new games as well, and we have implemented many such games, often in a matter of hours.

The second research objective is to provide the researcher in this field with an experimental environment in which research on parallel and distributed game-tree search can be done. Research areas include distributed search algorithms and improvements on heuristics that guide the search.

The usefulness of such an experimental research infrastructure is demonstrated by the fact that we were able to implement and evaluate a new parallel search strategy

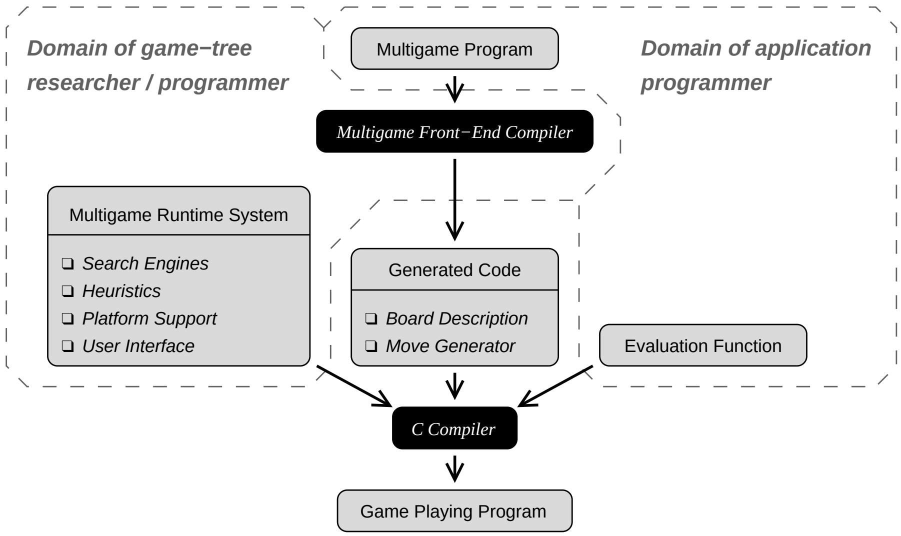


Figure 1.1: The structure of Multigame.

(see Chapter 5) in a few weeks, due to the modular and extendable design of the Multigame implementation. The new parallel search strategy increased the efficiency of some parallel search problems substantially. Depending on the application, the new strategy performs 1.5 to 11.8 times better than the best known traditional search strategy on a 128-processor distributed system, and achieves speedups that are close to linear.

1.1 Multigame

Multigame is designed to provide fast, parallel search on a distributed memory system. Its structure is based on that of traditional game playing programs: a move generator, a search engine, an evaluation function, and heuristics that guide the search. The first goal, providing the programmer with an easy programming model, is tackled by designing an application-domain-oriented language in which the rules of a game can be specified with little effort. The second goal, providing the researcher with an experimental environment, is addressed by designing the Multigame software in an extendable and modular way with clean interfaces between the modules. In this section we show how the Multigame system is organized.

Figure 1.1 shows the overall architecture of the Multigame system. Multigame consists of a *front-end compiler* and a *runtime system*. The front-end compiler generates game-specific code from the description of a game. The runtime system provides game-independent functionality and supports a broad class of games. From these components, the Multigame system generates a program that can play a game on a distributed system.

The figure shows the boundaries of the domain of the application programmer and that of the game-tree researcher. The game-tree researcher provides the application programmer with a system (compiler and runtime system) by which game-playing programs can be generated. The application programmer describes the rules of a game in the Multigame language. The primary task of the Multigame front-end compiler is to construct a move generator from a Multigame program. The move generator accepts the current board position as input and outputs all positions that can be reached by doing a legal move as defined by the rules. The generated code for the move generator is in ANSI-C, which is efficient and portable. The move generator runs sequentially, because the parallelism within the move generator is too fine-grained to be exploited efficiently on off-the-shelf distributed memory hardware.

The runtime system provides components that are common to many games, and is also written in ANSI-C. The application programmer selects which components are to be used. To create a fast executable, the source code of the runtime system is recompiled for each different game.

One of the most important components in the runtime system is the set of search engines. The runtime system uses a search engine that is suitable for the game; the application programmer can select another engine if desired. Parallelism is exploited within most search engines. Since the search engines used by Multigame are part of the runtime system, the knowledge about parallel execution is thus implemented in the runtime system. This is in contrast to parallelizing compilers, which analyze the program being compiled to discover parallelism.

The Multigame runtime system also contains search enhancements that are mostly independent of the game being played, such as iterative deepening, transposition tables, and history tables. The application programmer selects which search enhancements are used. The transposition table, a cache that stores search results for positions that have been searched, is for most games an important search enhancement. The Multigame runtime implements the transposition table efficiently for use on distributed memory systems.

Other components in the runtime system contain platform-specific code for a variety of platforms, including Linux, Solaris, Amoeba, and the Panda portability layer [9]. The runtime system also features a generic user interface, similar to a command shell. It accepts commands to play a game, set parameters, and print statistics.

An evaluation function is highly game-dependent. Ideally we would like the front-end compiler to generate appropriate evaluation functions from the rules of the game, but this is hard to do and is a (for that matter, interesting) research topic on its own. Pell [82, 83] and Epstein [45, 46] have developed methods for strategy planning, but these methods have not led to play on expert level for games of interest. In Multigame, the programmer can optionally provide an evaluation function (in C) for a specific game to improve the quality of the playing program. Without an evaluation function, the program will play the game poorly.

The C compiler translates the appropriate source files and links the objects to a program that can play the game in parallel.

The Multigame runtime system is a useful research environment for game-tree programmers. Since many games have been implemented on top of the Multigame runtime system, the efficiency of new search algorithms or heuristics can be tested on a variety of games. Also, the Multigame compiler is a suitable tool for research on a language to describe the rules of a game.

1.2 Contributions

This thesis presents a number of new ideas. We summarize the contributions for the board-game application programmer as follows:

- We show that an application-domain-oriented language and runtime system can hide parallelism from the application programmer. The programmer is freed from the burden of explicit parallel programming.

- We introduce the Multigame language, a language to describe the rules of a board game. A parallel game-playing program can be generated from a program written in this language. The language is easy to use and allows most board games to be expressed easily (see Chapter 3).

The contributions with respect to research on parallel and distributed game-tree search are the following:

- We provide the game-tree researcher with an experimental environment for doing research on distributed game-tree search. The modular structure of the Multigame runtime system makes it easy to experiment with new search techniques. The efficiency of new search techniques can be tested for a variety of games. The experimental environment contains highly optimized program code.

- We investigate application-specific network interface software, and show that optimizing network interface firmware is difficult but useful (see Section 4.4.4). We customized the firmware of the Myrinet network interface boards, so that it runs part of the application software on the network processor. This way, remote transposition-table lookups are serviced much faster than using general-purpose communication software.

- By studying prefetching techniques for shared transposition tables, we show that remote transposition table lookup latencies can be partially hidden (see Section 4.4.5).

- We present experimental performance comparisons between partitioned and replicated transposition tables. The results show that replicated transposition tables perform surprisingly well on systems of up to 32 processors when the network has a high bandwidth available (see Section 4.7). On larger systems, partitioned tables are more efficient, provided that the network latency is low.

- We present Transposition-Driven Scheduling (TDS), a new parallel scheduling algorithm for distributed IDA*. TDS combines the work distribution with a distributed transposition table. We show that TDS is both efficient *and* scalable (see Chapter 5); on a distributed memory system with 128 processors, we obtain efficiencies between 92.6% and 97.2%, where efficiencies of traditional implementations are between 6.4% and 61%.

1.3 Our experimental environment

Although Multigame runs on a variety of platforms, including shared memory multiprocessors, it is optimized to run on a distributed memory machine. Our initial implementation ran on a cluster of 80 SPARC Classic clones, each containing a 50 MHz MicroSPARC II processor and 32 MB RAM, and connected by a 10 Mbit/s partially switched Ethernet, using the Amoeba distributed operating system [114].

Later we switched to a system consisting of 128 computers. Each machine is equipped with a 200 MHz Pentium Pro processor with 256 KB full-speed, on-die, second level cache, and has 128 MB main memory. The machines run the Linux RedHat 5.2 operating system. We use this system, called DAS, for all measurements described in this thesis.

All machines are connected through 100 Mbit/s partially switched Ethernet, and through Myrinet [21], a 1.2 Gbit/s bi-directional switching network. The Ethernet is used as a control network for starting applications and sharing files over NFS. The applications communicate over Myrinet. The Myrinet network interfaces have a LANai 4.1 or LANai 4.3 RISC processor and 1 MB of SRAM memory. The Myrinet network consists of 32 8-port switches that are organized in a two-dimensional (4×8) grid with wrap-around. Each switch connects to four other switches and to four host machines.

Each network interface board contains a programmable network processor. The applications run on top of Panda [18], which in turn uses LFC [15, 17] as communication substrate. The combination provides high throughput and low latency communication, both for unicast and multicast messages. The applications directly communicate with the network interfaces without issuing system calls, saving expensive kernel/user-space context switches.

1.4 Outline of this thesis

This section gives an outline for the remainder of this thesis.

Chapter 2 begins with an explanation of game-tree search, both for single-agent and two-player game trees. We discuss five search algorithms: MiniMax, Alpha-Beta, NegaScout, and MTD(f) for two-player games, and IDA* for one-player games. Then, we explain the basics of parallel game-tree search, and show why game trees are hard to search efficiently in parallel, explaining the overheads that lead to sub-linear speedups. Chapter 2 also puts this work in the context of related work. We discuss a number of problem-solving environments for the application domain of game playing.

In Chapter 3, we introduce the Multigame language. Rather than giving a formal description of the language, we use several intuitive examples to explain the basics of the language. Programs written in this language are compiled to move generators by the Multigame front-end compiler. Since code generation is non-trivial for this language, Chapter 3 also describes how the code generated by the front-end compiler works. We discuss design decisions and analyze the strengths and weaknesses of the Multigame language as well.

Chapter 4 discusses the Multigame runtime system. We give an overview of the runtime system, discuss the parallel search engines, work distribution, various implementations of a distributed transposition table, network firmware optimizations for decreased remote transposition table lookup latencies, remote transposition table lookup prefetching, the history heuristic, quiescence search, position repetition detection, pattern databases, 15-puzzle heuristics, and the Multigame user interface for interactive game playing. Section 4.7 gives extensive performance results for the Multigame system for six different games. We describe our experiences with implementing a multi-threaded and distributed game-playing runtime system.

In Chapter 5, we describe *Transposition-Driven Scheduling*, a new parallel scheduling algorithm that efficiently combines IDA* search and a distributed transposition table. Previous parallelizations of the IDA* search algorithm either were suboptimal because of the absence of a transposition table, or scaled poorly because of the high communication overhead to share a transposition table. Transposition-Driven Scheduling achieves nearly linear speedups on large-size distributed memory systems. We explain the algorithm, describe related work, and discuss implementation issues. The performance for three one-player applications is compared in detail to traditional work-stealing based approaches. We also discuss future applicability of Transposition-Driven Scheduling for two-player search algorithms.

In Chapter 6, we draw overall conclusions. We evaluate to what extent we have reached our goals, and summarize the lessons learned. During the research for this thesis we came across some issues that still need to be worked out; therefore we list directions for future research.

Appendix A shows example Multigame programs for the 15-puzzle, connect-4, and chess. The test positions for the performance experiments for various applications are given in Appendix B. Appendix C lists the user interface commands for (interactive) game playing.

Chapter 2

Background and related work

In this chapter, we first explain the basics of game-tree search, both for one-player and two-player game trees. Section 2.1 can be skipped safely by people already familiar with game trees. In Section 2.2 we discuss parallel game-tree search, and explain why efficient parallel game-tree search is difficult. In Section 2.3 we discuss related work on problem-solving environments for game-playing programs.

2.1 A background on game trees

In this section we discuss the basics of game-tree search. We briefly describe the main search algorithms used in this thesis: Alpha-Beta, NegaScout, and MTD(f) for two-player games and IDA* for one-player games.

We will first explain the concept of a game tree. Whenever a game-playing program is to make a move, the program searches a so-called game tree to find the best move from the given board position. For one-player games (e.g., the sliding tile puzzle) the goal is to find the shortest sequence of moves from a problem instance to a target solution. For two-player games, the goal is to find the best move under the assumption that the opponent replies with the best countermove, subject to time constraints.

Game-tree search algorithms recursively analyze moves ahead. Each node in the tree (or rather: graph, as we will see later) corresponds to a position; in practice the terms *node*, *position*, and *state* are used interchangeably. Each edge in the tree corresponds to a possible move from that position. Most games of interest generate trees that are too large to be fully searched. Although good search algorithms will avoid searching uninteresting parts of the tree, game-tree search algorithms typically limit the depth of the tree. In game-tree jargon, a move from one position to another is called a *ply*. Non-leaf nodes are called *interior* nodes.

Figure 2.1 shows an example game tree for a two-player game. The numbers in the

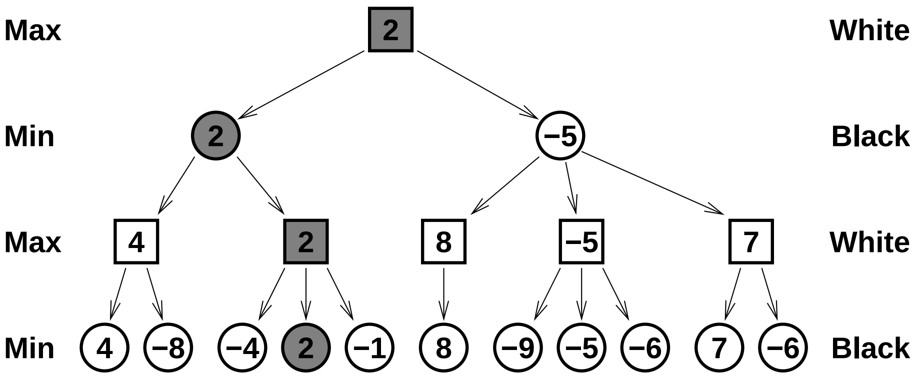


Figure 2.1: Example game tree.

leaves of the tree are *evaluation values* corresponding to each position; a high value indicates that the position is advantageous for the white player. White (represented by squares in the figure) tries to maximize the values of each of the children, while black (represented by circles) tries to minimize them. For example, the value of the root node is $\max(2, -5) = 2$; this value is called the *minimax* value of the tree. Evaluation values are usually integer values; search algorithms like MTD(f) even require the use of integers. The optimal line of play (the *principal variation*) is shaded gray in the figure.

Figure 2.2 shows pseudo code for the MiniMax search algorithm, that recursively determines the minimax value of a tree. It searches the tree up to a specified depth. If this depth is reached, or if the node is a terminal position (i.e. a win, draw, or loss), the *evaluation function* determines the evaluation value. For terminal positions, the evaluation function returns ∞ in case of a win for white, $-\infty$ when black wins, and 0 for draws.[1] Otherwise, the values of the children are maximized or minimized, depending on whether white or black is to make a move.

The pseudo code is a simplification of the real MiniMax algorithm.

In practice, one also wants to know *which* child is the best child. Therefore, the function constructs a principal variation as well.

The MiniMax search algorithm is inefficient for a number of reasons. One reason is that in unaltered form, the algorithm searches *trees*, rather than recognizing that the search space of most games are *graphs*. The name “game tree” is unfortunate, but is a historical legacy. A position that can be reached by different sequences of moves is called a *transposition*. An example of a transposition is given in Figure 2.3. Here, the chess opening moves e4–Nf6–d3 and d3–Nf6–e4 yield the same position. Searching the subtree below a transposition more than once wastes processor time.

[1]Some programs distinguish between a terminal 0 and a non-terminal 0, to postpone a draw as long as possible, in the hope that the opponent makes a mistake.

```
FUNCTION MiniMax(Node : NodeType; DepthToGo : INTEGER) : INTEGER
    IF DepthToGo = 0 OR Terminal(Node) THEN
        Value := Evaluate(Node);
    ELSIF WhiteToMove(Node) THEN
        Value := −INFINITY;
        FOR EACH Child IN Children(Node) DO
            Value := MAX(Value, MiniMax(Child, DepthToGo − 1));
        END
    ELSE
        Value := INFINITY;
        FOR EACH Child IN Children(Node) DO
            Value := MIN(Value, MiniMax(Child, DepthToGo − 1));
        END
    END

    RETURN Value;
END
```

Figure 2.2: The MiniMax search algorithm.

The *transposition table* [23, 110, 122] detects and exploits transpositions. The table serves as a large cache, storing search results of positions that have already been searched. Each time a position is to be searched, the transposition table is consulted to check whether the position has been searched previously. If this is the case, the cached search results are used, rather than searching the position again. We elaborate on transposition tables in Section 4.4.

A second reason why the MiniMax search algorithm of Figure 2.2 is inefficient, is that MiniMax searches subtrees that are irrelevant for determining the minimax value at the root position. Consider the example tree in Figure 2.4. The value of position c does not contribute to the value at the root. Since a is the minimum of b and c, a is at most 4. The minimax value of the tree is the maximum of 9 and the value of a, so the minimax value is always 9, independent of c. Therefore it is not necessary to search c. Intuitively, black's move from a to b is a refutation for white's move from the root to a. It is useless to search for more refutations (e.g., the move from a to c) as white will never play the move to a. In larger trees, entire subtrees can be pruned this way. Search algorithms like Alpha-Beta, as discussed below, are able to prune needless work.

Usually, search algorithms are not implemented with alternating min and max nodes. Instead, each node *maximizes* the *negated* value of its children. Effectively, the signs before the search results of the positions where the black player is to move are negated. The resulting algorithm is commonly referred to as NegaMax [66]. The tree searched by NegaMax and MiniMax is exactly the same, NegaMax is just simpler to implement. For this reason, we use negamax search trees in the remainder of this thesis.

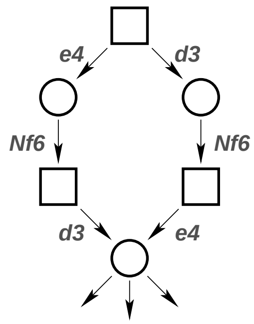


Figure 2.3: Example of a transposition.

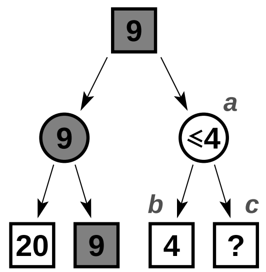


Figure 2.4: Opportunity to prune work.

In the examples in this section, we assume that the two-player game trees are searched up to a fixed depth. In practice, critical leaf nodes are searched deeper. We elaborate on this in the section on quiescence search (Section 4.5.2).

Most tree-searching programs hash positions to *signatures*. A signature is a large integer from an uniformly distributed space. Signatures can be used to distinguish positions: positions with different signatures are certainly different; positions with the same signature are likely the same. A signature is also used to index the transposition table.

An efficient method for computing a signature is described by Zobrist [122]. Each piece-field pair is associated with a random integer. These random numbers are stored in a matrix indexed by piece number and field number. The signature is obtained by taking the bitwise xor (exclusive or) of the random numbers associated with all pieces. This gives a uniformly distributed signature, suitable for hashing. The method is efficient, because the signature can be maintained incrementally. For example, when a single piece is moved across the board during a move, only two xor operations are required to compute the new signature: one for removing the piece from its source field and one for placing the piece at its destination field.

2.1.1 Alpha-Beta

The Alpha-Beta [66] algorithm is a well-known algorithm that prunes uninteresting parts of the tree as shown in the example of Figure 2.4. Figure 2.5 shows the pseudo code for AlphaBeta. The algorithm uses a *search window*, (α, β). The search window is used to specify the range within which results are interesting. Search results outside the search window are provably irrelevant (given the search depth constraint), either because the position is good for the white player and the black player has already found a path (a refutation) to avoid this position, or vice versa, as will become clear later.

```
FUNCTION AlphaBeta(Node               : NodeType;
                   Alpha, Beta, Depth : INTEGER) : INTEGER
    IF Depth = 0 OR Terminal(Node) THEN
        RETURN Evaluate(Node);
    END

    Value := −INFINITY;
    Child := FirstChild(Node);

    WHILE Child != NULL AND Alpha < Beta DO
        Value := MAX(Value, −AlphaBeta(Child, −Beta, −Alpha, Depth−1));
        Alpha := MAX(Alpha, Value);
        Child := NextBrother(Child);
    END

    RETURN Value;
END
```

Figure 2.5: The Alpha-Beta search algorithm.

The root position is searched with search bounds $(-\infty, \infty)$ to obtain the root's value. The function *AlphaBeta* works as follows. If the position is a leaf node, the evaluation value is returned. Otherwise, the children are searched one by one, recursively. The α and β bounds are exchanged and negated, since we search negamax trees. The search result can narrow the search window, by increasing α. As soon as α becomes greater or equal to β, it is proved that the current node is so good that the opponent will avoid this position. Since the position has become uninteresting, the remaining children are pruned.

Figure 2.6 shows an example that illustrates how Alpha-Beta works. The top (α, β) value for each node is the original search window for that node; the other (α, β) values show how the windows are narrowed after their first children were searched. First, node a, the root node, is visited with search bounds $(-\infty, \infty)$. Next, node b is searched, also with bounds $(-\infty, \infty)$. Then, node c is searched, which is a leaf node. The evaluation value for c, 20, is returned to b, and negated. The search bounds of b are narrowed to $(-20, \infty)$. Now d is visited with search bounds $(-\infty, 20)$. The evaluation value of d, 9, is returned, and this determines the final search result for b, which is -9. This result is reported to a, who adjusts its window to $(9, \infty)$. Then, e is searched with bounds $(-\infty, -9)$. Next, f is visited with bounds $(9, \infty)$. The evaluation value, 4, is reported to e, who negates the result and sets it window to $(-4, -9)$. Since e's α has surpassed its β, the remaining child g is pruned.

The result returned by the function *AlphaBeta* should be interpreted as follows. If the result is greater than the original α (the actual argument of the function) and smaller than β, the result equals the negamax value (this is the "real" value of the node). If the result is greater or equal to β, the negamax value is at least the result,

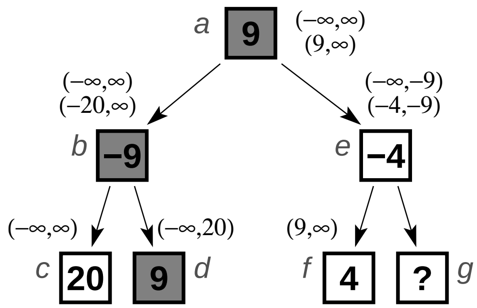


Figure 2.6: Alpha-Beta in action.

thus the real value may be even better than what was returned. The associated move from this position may not be the best move, but it was good enough to prove that the position is so good that the opponent will avoid it. If the result is smaller or equal to α, the negamax value is at most the result; however, the position is so bad that the player will avoid it anyway.

For simplicity, Figure 2.5 does not show the interaction with the transposition table. The transposition table stores the search results for Alpha-Beta. Since results do not always fall within the original (α,β) window, search results may be lower or upper bounds. Therefore, each result stored in the transposition table is tagged with an indicator that states whether the result is a lower bound, an upper bound, or exact. If the cached value is looked up at a later time, the "bound" tag is used to check whether the value is usable within the new (α,β) window. The transposition table is discussed in more detail in Section 4.4.

The Alpha-Beta algorithm is most efficient when as much work as possible is pruned. This depends on the order in which the children are searched. Alpha-Beta prunes most work when at each the best child is searched first at each interior node. When each best child is searched first, the maximum amount of work is pruned, and the algorithm searches the *minimal tree*. Unfortunately, it is not known in advance which child is the best child (if it were known, there is no need to search), so the algorithm relies on ordering heuristics to arrange the children in such a way that the most promising children are searched first.

Many ordering heuristics exist. An important one is the transposition table (also used for detection of transpositions), in combination with *iterative deepening* [110]. With iterative deepening, the root is searched to increasing search depths, starting from depth 1. Common increment steps are one or two plies. Whenever a node is searched, the node is looked up in the transposition table to see whether it was searched in the previous iteration. If this is the case, the best child from the previous iteration (or the child that was good enough to generate a cutoff), is searched first in the current

```
FUNCTION NegaScout(Node: NodeType;
                   Alpha, Beta, Depth: INTEGER) : INTEGER
    IF Depth = 0 OR Terminal(Node) THEN
        RETURN Evaluate(Node);
    END

    Child := FirstChild(Node);
    Best  := -NegaScout(Child, -Beta, -Alpha, Depth-1);
    Alpha := MAX(Alpha, Best);
    Child := NextBrother(Child);

    WHILE Alpha < Beta AND Child != NULL DO
        Value := -NegaScout(Child, -Alpha-1, -Alpha, Depth-1);

        IF Value > Alpha AND Value < Beta THEN
            Value := -NegaScout(Child, -Beta, -Value, Depth-1);
        END

        Best  := MAX(Best,  Value);
        Alpha := MAX(Alpha, Value);
        Child := NextBrother(Child);
    END

    RETURN Best;
END
```

Figure 2.7: The NegaScout search algorithm.

iteration, since it is a good candidate to be the best child again. Another important move-ordering heuristic, the *history heuristic*, which heuristically orders *all* children, is discussed in Section 4.5.1.

Iterative deepening can also be used to implement time control [108]. Based on the time needed to search an iteration, the program can estimate the time needed to search one ply deeper. A program can decide whether or not to start a new iteration, and even if such an iteration turns out to take too long, the program has a best move available from the previous iteration.

2.1.2 NegaScout

The NegaScout [93, 95] search algorithm is an enhancement of the Alpha-Beta algorithm. It is probably the most often used search algorithm in game-playing programs.

Like Alpha-Beta, NegaScout assumes that the search tree is strongly ordered (i.e., children are ordered in such a way that the most promising child is searched first). However, NegaScout uses different search bounds for all children except the first. The first child (the most promising one) is searched with the normal search bounds $(-\beta, -\alpha)$, and establishes an α in the parent node, that is not likely to change during

```
FUNCTION MTDF(Root : NodeType; Depth : INTEGER) : INTEGER
    LowerBound := -INFINITY;
    UpperBound :=  INFINITY;
    Pivot      :=  0;

    WHILE LowerBound < UpperBound DO
        Value := AlphaBeta(Root, Pivot-1, Pivot, Depth);

        IF Value < Pivot THEN
            UpperBound := Value;
            Pivot      := Value;
        ELSE
            LowerBound := Value;
            Pivot      := Value + 1;
        END
    END

    RETURN Value;
END
```

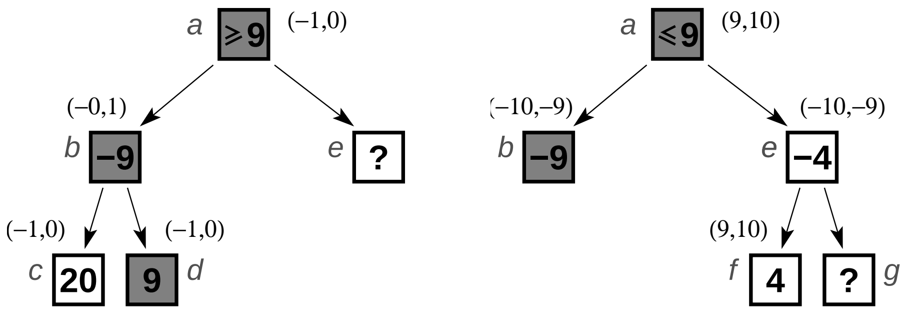
Figure 2.8: The MTD(f) search algorithm.

the search of the remaining children (if the first child did not immediately cause a cutoff, of course). All that has to be proved, is that the remaining children are inferior to the first one. Therefore, the remaining children are searched with the *minimal window* $(-\alpha-1,-\alpha)$. A minimal window search is cheaper than a full window search, since more work is pruned. If the negated search result is smaller than α (as is usually the case), the child is inferior and the next child is tested. Otherwise, we know that the child is superior and search it again, but now with the full window $(-\beta,-value)$, where *value* is the value returned by the minimal window search. The search result of the full window narrows the parent's (α,β) window and might cause a cutoff.

Figure 2.7 shows the NegaScout algorithm in pseudo code. The algorithm first checks whether the node is a leaf; in this case the evaluation value is returned. Otherwise, the first child is searched with the full window. As long as there are children left and α remains smaller than β, each child is first tested with a minimal window, and, if the child is better than the previous children, searched again with a full window. The search window is adjusted after the child is searched. Finally, the search result for the best child is returned.

2.1.3 MTD(f)

MTD(f) [87] pushes the idea of minimal window searches to the extreme. *Every* node is searched with a minimal window, even the root position. The result r of a search around a $(\beta-1,\beta)$ window is interpreted as follows: if $r \geq \beta$ then the node's value is greater or equal to r, otherwise the node's value is smaller or equal to r.



(a) First iteration. (b) Second iteration.

Figure 2.9: MTD(f) in action.

Multiple searches of the root position with different windows are needed to compute its negamax value. Figure 2.8 shows how this is done. The search starts around an arbitrary chosen value (when used in combination with iterative deepening, it is more efficient to start the search around the negamax value of the previous iteration). The first search either sets a lower or an upper bound on the root's value. Consecutive searches improve the lower and upper bounds, until both bounds equal each other, in which case the root's value is determined.

Figure 2.9 shows how MTD(f) searches the same tree as shown in Figure 2.6. In the first iteration, the root is searched with window $(-1,0)$. Via nodes b, c, and d, the value of b is determined, and equals -9. Since the root's β is 0, the tree below e is pruned. Now the lower bound on the negamax value of a is set to 9, because the result is greater or equal to the β of the search window. In the second iteration, the root is searched with window $(9,10)$. Node b is visited, but is found in the transposition table, since it was adequately searched in the first iteration. MTD(f) relies heavily on the transposition table to prevent searching the same subtree multiple times. Node e and f are searched, but g is pruned. Now the upper bound on the negamax value of a is set to 9, because the result is smaller or equal to the α of the search window. The true negamax value for a thus is 9, because 9 is both a lower and an upper bound.

In this simple example, MTD(f) searches more nodes than Alpha-Beta (a and b are visited twice). However, on large trees MTD(f) is much more efficient than Alpha-Beta, and on average somewhat faster than NegaScout [86]. The narrow search windows of MTD(f) and NegaScout cause many cutoffs, which make these search algorithms efficient.

```
FUNCTION Search(Node : NodeType; Bound : INTEGER) : INTEGER
    MinDist := Evaluate(Node);

    IF MinDist <= Bound AND NOT Terminal(Node) THEN
        MinDist := INFINITY;
        Child   := FirstChild(Node);

        REPEAT
            MinDist := MIN(MinDist, Search(Child, Bound – 1) + 1);
            Child   := NextBrother(Child);
        UNTIL Child = NULL OR MinDist = Bound;
    END

    RETURN MinDist;
END

FUNCTION IDA(Root : NodeType) : INTEGER
    NewBound := Evaluate(Root);

    REPEAT
        OldBound := NewBound;
        NewBound := Search(Root, NewBound);
    UNTIL OldBound = NewBound;

    RETURN NewBound;
END
```

Figure 2.10: The IDA* search algorithm.

2.1.4 IDA*

Iterative Deepening A* (IDA*) [67] is used for searching *one-player* games like the sliding tile puzzle and Rubik's cube. The objective is to find the shortest solution path from a given problem position to a target position (or one out of a number of target positions). IDA* is a memory-efficient variant of A* [79].

IDA* repeatedly descends the search tree, as shown in the pseudo code in Figure 2.10. Each iteration the tree is searched with an increased *search bound* (not to be confused with the (α, β) search window in Alpha-Beta search) that searches the tree more deeply, until a solution is found. The search bound is used to put a limit on the solution length: as soon as it can be proven that from the current node no solution is possible within the given bound, the subtree below the node is pruned. If the function returns the same value as the search bound argument, it has found a solution; if the return value is greater than the search bound, no solution is found and the return value is used as a new search bound for a subsequent iteration.

In one-player games the evaluation function is used to estimate the distance from the current position to a target position. If the distance exceeds the search bound,

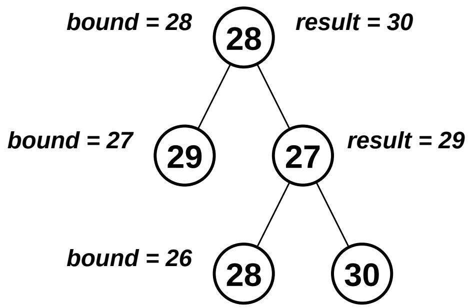


Figure 2.11: Example IDA* search tree.

all children are pruned and the distance is returned. If it never overestimates the distance, IDA* will find the shortest solution(s). Such an evaluation function is said to be *admissible*. A well-known example of an admissible evaluation function is the Manhattan distance in the sliding tile puzzle, which sums the distances between each tile's current position and the tile's target position. To prune as much work as possible, the evaluation function should estimate the minimum solution length as accurately as possible, but may not overestimate the solution length if minimal solutions are desired.

An example of an IDA* iteration is shown in Figure 2.11. The numbers inside the circles represent evaluation values. The tree is traversed depth first, left to right, with the search bounds as indicated. A cutoff occurs at each of the leaf nodes; here the evaluation value exceeds the search bound. The search result of the root equals 30. Since the search result of the root is greater than its search bound, the algorithm will start an new iteration with search bound 30; this tree will be deeper as the one shown in the figure.

2.2 Parallel game-tree search

A game-playing program needs to decide upon the best move under real-time constraints, and limits the search depth to keep the response time within reasonable bounds. A parallel game-playing program combines the processing power of multiple processors to extend the search a few plies, under the same time constraints. The underlying assumption is that deeper searches yield better play, although increasing the search depth eventually leads to diminishing returns [63, 64, 105]. Nevertheless, many parallel game-playing programs (for example, Deep Blue [56, 57], CilkChess (the successor of ⋆Socrates [62]), and Chinook [103]) have proven the merits of parallel search. Since game trees grow exponentially, searching one ply deeper requires a factor more processing time.

One-player search algorithms are not usually run in real time, but search for an

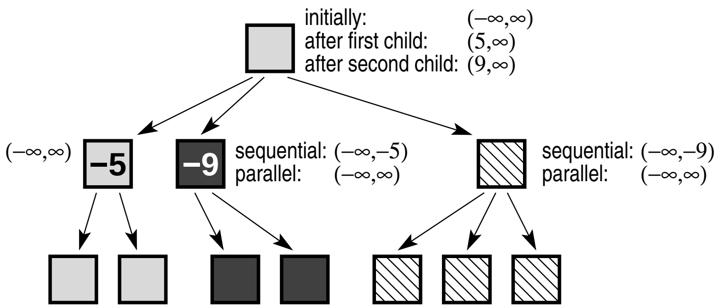


Figure 2.12: Parallel decomposition of a search tree.

optimal solution path. Here, the goal of parallel search is to decrease the time to find the optimal solutions for hard problems.

There are several ways to search trees in parallel. Most methods decompose the search tree and let each processor search part of the tree, as illustrated by Figure 2.12. One processor searches the light gray nodes; another processor searches the dark nodes, and a third processor searches the shaded nodes (the search windows next to the nodes are explained later).

2.2.1 Parallel two-player game-tree search

Two-player game trees are difficult to search in parallel [76], since Alpha-Beta-like search algorithms (including NegaScout and MTD(f)) suffer from several kinds of overheads: *search overhead*, *synchronization overhead*, and *communication overhead*. We will now discuss the causes of these overheads.

The Alpha-Beta search algorithm is inherently sequential. Since the (α,β) search argument of a child depends on the window of its parent, and because the window of the parent can be narrowed by the search result of each of its children, data dependencies exist throughout the search tree. To search the tree in parallel, these dependencies must be broken by *speculating* on search windows.

To understand the consequences of speculation, we use Figure 2.12 to show the difference in search bounds between sequential and parallel search. When the example tree of Figure 2.12 is searched *sequentially*, the search result for the root's first child narrows the root's search window to $(5,\infty)$. The second child will be searched with the narrowed search bounds $(-\infty,-5)$. The second child's search result narrows the root's search window to $(9,\infty)$, and the third child is searched with window $(-\infty,-9)$. If the first child would have caused a cutoff, the second and third child would not have been searched at all.

If the tree is searched in *parallel*, the search result of the first child is not available

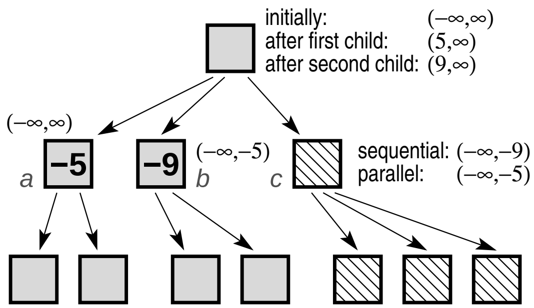


Figure 2.13: Young Brothers Wait.

when the search for the second and third child starts. Therefore, the search windows for the second and third child are conservatively set to the same window as the first child. In the example, the search windows are $(-\infty, \infty)$, and are wider than during sequential search. As a consequence, the parallel search will prune less work in the subtree of the second and third child than the sequential algorithm would have done.

With respect to sequential search, the extra amount of work performed by the speculative, parallel search is called *search overhead*. Search overhead might lead to a significant amount of extra work; on large-scale systems overheads of a factor two or three are typical. Consequently, the maximum obtainable efficiency is typically less than 50% [20, 27, 72]. Only Feldmann reports lower search overheads [48], but these performance numbers are criticized for a number of reasons [27]; in particular the slow processors in the test environment and the unusual amount of time spent for move ordering are held responsible for the high speedups.

Several optimizations reduce the search overhead. The Young Brothers Wait Concept (best described in Feldmann's thesis [48]) waits until the search result of the first child is established before the remaining children may be searched in parallel. Figure 2.13 illustrates how the Young Brothers Wait Concept works, using the same search tree as in Figure 2.12. The search result of a sets the α in the parent's search window. b and c may be searched in parallel, each with a $(-\infty, -5)$ window. Due to the move-ordering heuristics, the first child often is the best child. The first child establishes an α in the parent's search window that will not be improved by the remaining children if the first child indeed is the best child, and may even prune the remaining children. In the example this is not the case, since b turns out to be better than a. If the move-ordering heuristics would have correctly predicted b to be superior to a, b would have been searched first with window $(-\infty, \infty)$. a and c would have been searched later with the smaller search window $(-\infty, 9)$, the same as sequential search would have used. Thus, search overhead occurs when the move-ordering heuristics fail. Nevertheless, the Young Brothers Wait Concept reduces the search overhead with a significant amount. Paradoxically, the intentional reduction of parallelism improves the parallel search performance.

Another optimization propagates new search bounds as soon as they become available. When the second and the third child in the tree of Figure 2.12 are searched in parallel, both start with bounds $(-\infty, -5)$. If the search of the second child finishes before the search of the third child, the improved search bound $(-\infty, -9)$ can be propagated downward in the subtree below the third child, so that more work can be pruned. Similarly, if the third child finishes before the second child and improves the $(-\infty, -5)$ search bound, the new search bound can be propagated in the second child's subtree. This optimization is discussed in Section 4.2.2.6.

Apart from search overhead, parallel search suffers from *synchronization overhead*. Synchronization overhead is the result of idling processors that wait for search results to become available. To reduce the synchronization overhead, it is possible to let idle processors start work on other parts of the tree, if such work is available. This is not always the case, since the Young Brothers Wait Concept reduces the amount of parallelism.

On distributed memory systems the exchange of data leads to *communication overhead*. Communication overhead can significantly reduce parallel performance. Communication is necessary for several purposes. First, for many games the transposition table works much better if the table is shared between the processors, and keeping the tables consistent requires much communication. In Section 4.4 we elaborate on shared transposition tables. Second, the work has to be distributed over the processors, which also requires communication.

On distributed memory systems, work distribution can be implemented using work stealing. Each processor maintains a local job queue. Whenever a processor creates work, the work is appended to the local queue. If a processor needs work, it tries to dequeue a job from the local queue. If the local queue is empty, the processor randomly selects another processor and tries to steal work from the selected processor's queue. Section 4.3 elaborates on work stealing.

An example of a work-stealing parallel search algorithm is Jamboree search [62, 71], which is a parallel variant of NegaScout. The first child is searched (with a full window) before the remaining children are searched (with a minimal window). Null-window searches of the remaining children may proceed in parallel. To reduce the search overhead, a full window re-search cannot be performed before the elder brothers (i.e., the more promising brothers) have completed their minimal-window searches. Moreover, possible full window re-searches on different children are performed one after another. Therefore, the expensive full-window researches are never searched speculatively, and this reduces the parallel search overhead.

There are other ways to distribute the work over the processors. In Chapter 5 we discuss *Transposition Driven Scheduling*. A number of other solutions appeared in the literature. Brockington gives an overview of parallel search algorithms [26]; we discuss a few algorithms below.

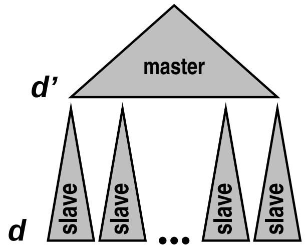


Figure 2.14: An APHID search tree.

The APHID search algorithm [27, 29, 30] parallelizes Alpha-Beta and is suitable for execution on shared memory and distributed memory machines. The algorithm was designed to allow easy integration with existing Alpha-Beta search programs. A master process repeatedly searches the top of the tree up to depth d', while slave processes independently search the subtrees below depth d' (see Figure 2.14). The slaves search the subtrees in increasing steps up to depth d. A node at depth d' uses an estimated value and is marked as *uncertain*, until the slave process has fully searched the node up to depth d. The master process traverses the top level tree until all nodes at depth d' are marked as *certain*. When the master visits a depth d' node for the first time, it assigns a slave processor. Multiple slaves are assigned to one processor to balance the load. Once assigned, the node never moves to another processor, thus the subtree below the node is searched to increasing depths by the same processor. The slaves use a non-shared transposition table. When a depth d node is searched to an increased depth, the move-ordering information obtained during the previous iteration can be found in the local transposition table, since a subtree is always searched by the same processor. The most important transposition-table entries (for nodes in the top of the search tree) are shared so that top-level transpositions are detected. Transpositions at lower levels are not discovered if searched by different processors. Reported speedups for APHID range from 14.4 for checkers to 18 for chess and 37 for Othello on a 64-processor shared memory SGI Origin 2000.

The ABDADA search algorithm [121] uses a shared transposition table to parallelize Alpha-Beta. All processors start searching the root simultaneously. Each entry in the table is extended with a field that counts how many processors are busy on the associated node. This entry is used to select the order in which the children of a node are searched. Three criteria determine the parallel search order. The first child is searched, regardless of how many processors are already searching the child. Then, the remaining children not being searched by other processors are visited. Finally, the children which have not been searched completely are analyzed to help the other processors that search these children.

ABDADA is hard to implement efficiently on distributed memory systems, since it updates the shared transposition table very frequently. Another disadvantage of ABDADA is that it does not propagate updated search bounds downward in subtrees that are being searched, although the algorithm could be extended to do so. ABDADA achieves a speedup of 16 on 32 processors for chess.

2.2.2 Parallel one-player game-tree search

One-player game trees are simpler to search in parallel than two-player game trees, when the IDA* search algorithm is used to search the trees. The reason for this is that there are no data-dependencies between the children of a node within an iteration; therefore no speculation is needed to obtain parallelism. Only the iterations are synchronization points. A simple strategy that scales nearly linearly is presented by Rao et al. [92]. The search tree is partitioned and distributed over the available processors. Rather than eagerly distributing the subtrees, work stealing can be used to lazily distribute the game tree. In this case, an idle processor can help a busy one by taking over part of its work.

Another way to parallelize IDA* is Parallel Window Search [89]. PWS independently searches the same tree on different processors, but each processor uses a different search bound. The parallelism is speculative. Some processors may search the tree with a search bound that is too high; this work is wasted. Sequential IDA* never searches a tree with a bound that is too high, therefore PWS scales poorly. Moreover, the load is badly balanced, since a tree with a low search bound requires much less time to search than a tree with a high search bound.

Although IDA* does not use the transposition table for move ordering, the transposition table is useful to detect transpositions [94]. As far as we know, nobody has combined search enhancements like the transposition table with parallel IDA*. Although work stealing IDA* itself yields near-perfect speedups, the speedups drop drastically when a shared transposition table is added (albeit compared to a better sequential algorithm), due to the increase in communication overhead. In Chapter 5 we present an alternative to work stealing, called Transposition Driven Scheduling, which efficiently combines parallel IDA* and a shared transposition table.

2.3 Problem-solving environments for generic game-playing programs

A *problem-solving environment* (PSE) is a system that provides the user with an infrastructure to solve problems within a particular class of applications. PSEs exist for many application domains: spreadsheets, word processors, and calculators are commonly used in daily life. Software developers often use a PSE like *make* to generate executable programs from source code. Engineers design complex objects using AutoCAD. PSEs are also used in computational science: examples include Matlab for solving mathematical problems, and SPSS for statistical data processing. Yet another example is CTADEL [44], a code generating system for solving partial differential equation based problems. In his thesis [44], van Engelen gives an overview of PSEs for scientific computing.

The interface of a PSE is of a high level, natural for the application domain, and hides the details of the hardware the system is running on. Most of these interfaces are graphical. However, some systems use a high level language that is specifically designed for the application domain. The programmer can easily describe a "problem" in such a language. These programming languages are less flexible than general-purpose programming languages, but offer a simpler programming model.

Several computationally intensive PSEs are able to generate solutions that can exploit parallelism. These systems exploit parallelism implicitly and use knowledge about their application domain to decompose the work in parts that can be processed in parallel. They free the programmer from the burden of explicit parallel programming.

In this section, we will focus on PSEs for the domain of game playing. Several PSEs for this domain have appeared in the literature. Below, we discuss *DIB*, *Smart Game Board*, *Hoyle*, *Metagame*, and *SearchBench*, and their similarities and differences with Multigame.

DIB — Distributed Implementation of Backtracking

Distributed Implementation of Backtracking (DIB) [50] is an early general-purpose package for programming parallel search problems on a distributed memory machine. DIB is a set of library routines that provides the basic support for exhaustive search problems, branch-and-bound, and Alpha-Beta search. DIB is not a mature problem-solving environment, but we include it here because it facilitates implementing distributed search programs and because it is probably the primary PSE for search.

The programmer must provide some call-back functions so that the library can construct the search tree. One of those user-supplied functions is used to generate children from a given parent; another function updates the parent's state when one of its children finishes.

DIB uses work stealing to distribute the work. Fault tolerance is implicitly obtained by redoing work when processors would otherwise be idle.

The performance results presented for Alpha-Beta search are poor: at six processors the speedup for searching an uniformly distributed search tree tapers off to a factor 2.8. It is not clear whether the application uses a transposition table or other move-ordering heuristics. The application uses DIB's "update children" mechanism to propagate new (α,β) search bounds. Speedups for branch-and-bound applications are much better, and are excellent for exhaustive search problems.

Smart Game Board

One of the earlier attempts to build a generic tool for playing board games was Smart Game Board [65]. Although the environment was designed to support the entire class of two-player, perfect information board games, the research mainly focussed on the development of a state-of-the-art go-playing program. In that time (the late 1980s), computer go was in its infancy, and go playing programs were hardly able to beat even novice human players.

Smart Game Board is a case study in knowledge and software engineering. At the start of the project, the requirements, feasibility, programming effort, and the ultimate playing level of a go-playing program were hard to predict, and from the beginning it was clear that the software would undergo major changes as more experience was obtained. This made the domain of game-playing a challenging testbed for research in software and knowledge design and engineering. Smart Game Board was successfully used to (partially) implement go, go-moku, Othello, chess, and nine men's morris. Go and Othello are the only games that could actually be played. The other games were only supported by the largely game-independent graphical and interactive tool to maintain game databases. The tool could be used by human players to analyze games.

The differences with Multigame are apparent. Multigame is designed to play one and two-person games using distributed search algorithms. Although go falls inside the class of board games with perfect information, the state space of go is too large to be searched by MiniMax search algorithms, rendering go an impractical application for Multigame. Go Explorer, the Smart Game Board implementation of go, does not search, but uses game-specific knowledge to select the supposedly best move from a position.

Hoyle

Hoyle [45] is another environment for two-person, perfect information board games. Unlike Smart Game Board, this environment is a testbed for machine learning. Traditionally, game-playing programs rely on deep searches of the game tree and on a game-dependent evaluation function. Human masters search less, learn from experience, and still make the right decisions. Hoyle is able to learn how to play a game, based on the experience obtained in previously played games, preferably against experts. Rather than using search as a base for move selection, Hoyle consults *advisors* when it is to make a move. Each advisor comments on all legal moves, recommending or rejecting each particular move. For example, the advisor "not again" votes against a particular move if it recognizes the move as the one that led to a loss in a previously played game. None of the advisors looks more than two plies ahead. Knowledge about longer move sequences is obtained by learning from previous games. The amount of game-specific knowledge used by Hoyle is minimal.

Performance studies on simple games showed that Hoyle indeed learns to play a game well. Epstein also describes how a pattern oriented advisor extends the learning capabilities of Hoyle [46]. Hoyle does not perform well for complex games like chess.

Hoyle differs from Multigame in important respects. Multigame uses search to select the best move and needs a game-specific evaluation function to guide the search. Although the learning capabilities of Hoyle relieve the programmer of the burden of writing an evaluation function, we feel that for current games of interest, programmed human knowledge is indispensable for playing at an expert level.

Metagame

There are many game-specific programs that play one game outstandingly, such as Deep Blue [57] in chess, Chinook [106] in checkers, and Logistello [32] in Othello. These programs carry much game-specific knowledge. This knowledge is obtained by a human expert and translated into an evaluation function. Focusing on one particular kind of game has several disadvantages. Each time a program for a new game is developed, significant human effort is required to analyze the game. The strength of the program depends on the knowledge level of the human expert. Moreover, the research results obtained for one particular game are sometimes hard to generalize and apply to other games.

From a researcher's perspective it is interesting to automate the process of learning to understand how to play a game well. This led to the introduction of Metagame [82], a framework for two-player, chess-like games. Like Hoyle, Metagame provides an environment for machine learning, but Hoyle learns from experience, while Metagame does not enforce this.

The rules of a game are expressed in a language that is designed for that purpose. The idea is to let a Metagame-playing program analyze the rules of the game so that it can plan a strategy to play the game. One novel feature of the system is that it can generate a random game within the class of Metagame games. A tournament can be held where all contestants (Metagame-playing programs) have 24 hours to analyze the rules of such a random game. The contestants play a prespecified number of games against each other; the objective is to win as many contests as possible.

Metagamer [83] is a program that plays Metagame games. Unlike Hoyle, Metagamer relies on an evaluation function and Alpha-Beta search to select moves. The evaluation function covers general game-independent features like material value and mobility, although the weights used for each of the features are different for each game.

A resemblance between Multigame and Metagame is the fact that both environments use an application-domain-oriented language to describe the rules of a game. The Multigame language is more expressive than the Metagame language (for example, en-passant moves and castling in chess cannot be expressed in Metagame), but expressiveness was not a real issue in the Metagame research. The class of games that can be expressed in Metagame is a subset of the class of games that can be expressed in Multigame, and it is possible to write a program that translates a Metagame program to a Multigame program.

Multigame relies on a game-specific evaluation function; in practice these evaluation functions are programmed by humans. It is possible to generate an evaluation function automatically by analyzing the rules of a game, although such an evaluation function is likely to lead to weaker play than an evaluation function that carries human knowledge. The Metagame language is more suitable for automatic analysis than the Multigame language, because the Metagame language is designed for this purpose, and the Multigame language is designed for expressiveness. For example, it would be hard to recognize capture moves in a Multigame program at compile time, while the Metagame language explicitly distinguishes between capture and non-capture moves [84]. Ultimately, a program could analyze a Metagame program, creating an equivalent Multigame move generator plus an evaluation function, so that the game could be played in a Multigame environment.

SearchBench

SearchBench [52] is a tool for exhaustive backward search. The tool is used to create, verify, and correct databases (due to software and hardware errors). The databases are created using retrograde analysis. The databases can be combined with forward search algorithms like Alpha-Beta for two-player games and IDA* for one-player games. Forward search uses little memory, although a transposition table is necessary to detect states that are encountered multiple times. Forward search is selective: uninteresting parts of the state space are pruned by the algorithm. In contrast, backward search is exhaustive within a preselected part of the state space. Backward search requires a large amount of memory, but is computationally efficient.

SearchBench was used to prove that nine man's morris ends in a draw when both players play perfectly. SearchBench was also used for the 15-puzzle. The databases created for the 15-puzzle significantly reduced 15-puzzle search times (Culberson and Schaeffer were the first who published this result [35–37]). The databases were also used to make progress in finding the hardest 15-puzzle positions; 13 positions with solution length 80 were found.

SearchBench differs considerably from Multigame. SearchBench was designed as a problem-solving environment for backward search, while Multigame was designed to play games on a distributed system. Multigame cannot be used for backward search, although Multigame can use pattern and endgame databases created by a separate program. SearchBench was not designed to run on a parallel or distributed system. To specify the rules of a game, Multigame uses a language that is specifically designed for this purpose. With SearchBench, the programmer must provide the system with a Pascal function to generate all possible forward moves from a position, and one to generate all possible backward moves from a position.

	DIB	Smart Game Board	Hoyle	Metagame	Search-Bench	Multigame
primary design goal	distri-buted search	knowledge and software engineering	planning by expe-rience	planning by rule analysis	bidirec-tional search	parallel and distributed search
nr. players	1 or 2	2	2	2	1 or 2	1 or 2
language for rules	–	–	–	$\surd$	–	$\surd$
parallelism	$\surd$	–	–	–	–	$\surd$
learning	–	–	$\surd$	$\surd$	–	–

Table 2.1: Properties of general game-playing environments.

Summary

Table 2.1 summarizes the most important aspects of the generic game-playing environments. Clearly, Multigame distinguishes itself with its ability to play a game on a parallel or distributed machine; only DIB provides rudimentary support for distributed search, but its infrastructure is too simple to search game trees efficiently. Hoyle and Metagame focus on machine learning, while SearchBench is used as a tool for backward search.

Chapter 3

The Multigame language

In this chapter we discuss the design and the implementation of the Multigame language. Multigame is an application-domain-oriented, high-level language for describing board games. A Multigame program describes the legal moves of a game in a formal way. The Multigame front-end compiler uses this description to generate a move generator, which is then linked with the Multigame library (see also Figure 1.1). The language offers the programmer a simple programming model. Moreover, the language hides parallel programming issues such as communication, synchronization, work and data distribution, and deadlock prevention from the programmer.

The outline of this chapter is as follows. First, we sketch the class of problems that can be handled by Multigame. Next, we describe the basics of the Multigame language. Then, we discuss the implementation of the Multigame front-end compiler. We give performance results for the move generators constructed by the front-end compiler. Although evaluation functions are not part of the Multigame language itself, we also discuss their role here, because the programmer has to provide an evaluation function as well as a Multigame program. We conclude with an extensive discussion. Parts of this chapter were published in [97] and in [98].

3.1 Design issues of the Multigame language

As with any application-domain-oriented language, the class of problems that can be expressed in Multigame is restricted. To keep the Multigame system manageable, we also restrict the kind of games that can be played with Multigame. Relaxing one of the restrictions listed below would complicate both the language and the implementation of the compiler and runtime system. The restrictions are the following:

- The language is designed for board games only. Concepts like *boards*, *pieces*, *fields*, *players*, and *moves* are embedded in the language. Card games (like bridge) and computer games (like Tetris and Quake) cannot be expressed, since they are not based on these concepts.

- The game must be either a one-player game (e.g., the 15-puzzle) or a two-player game (e.g., chess, checkers, Othello, tic-tac-toe).[1]
- The board consists of a rectangular, two-dimensional grid of fields. Each field holds at most one piece. Not all fields have to be used (this is useful for games like checkers, nine men's morris, and Pegged). The programmer can unfold a three-dimensional "board" (as used with Rubik's cube) to a two-dimensional representation (as illustrated by the figures in Appendix B.1).
- Perfect information is required. This excludes games such as Stratego where one cannot see the identity of the opponent's pieces.
- Finally, we exclude games that depend on chance or probability. Therefore, it is not possible to play backgammon and Risk, because they require dice.

By conforming to these restrictions, we have created a problem domain for which problems can be solved using parallel implementations of well-known search strategies like Alpha-Beta search [66], NegaScout [93, 95], MTD(f) [87], and IDA* [67] (see also Sections 2.1 and 4.2).

3.2 Principles of the Multigame language

This section explains the basics of the Multigame language. Rather than supplying a formal language definition, we give an intuitive description on the basis of some sample code fragments. The Multigame reference manual [99] describes the language in detail. Complete Multigame programs of the 15-puzzle, Connect-4, and chess are given in Appendix A.

A Multigame program is a formal definition of the rules of a game. The program specifies the number of players, the geometry of the board, the pieces, and the legal moves, each in a different section.

Figure 3.1 shows a complete Multigame program for tic-tac-toe. Names in boldface are keywords in the Multigame language; all other identifiers are user-defined. The first line specifies that the game is played on a 3×3 board. The ***pieces*** declaration declares one piece, named *mark*, which is represented as an 'X' for the white player and an 'O' for the black player.

The remainder of the program describes the rules for legal moves, using a combination of the Logo [81, 111] and Prolog programming paradigms. To describe a move, Multigame uses several implicit variables:

[1] The game of life can be expressed, although strictly speaking it is a zero-player game, since the sequence of moves from the starting position is fixed. It can be implemented as a one or two-player game, for which a player always has the "choice" of a single move.

```
dimensions (3,3)

pieces
{
    mark          'X' 'O'
}

main =            try new_mark else draw.

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

three_in_a_row =  find own mark,   # start from any position
                  any direction,
                  repeat 2 times [ step, points at own mark ].
```

Figure 3.1: A Multigame program for *tic-tac-toe*.

- There is a *current board state*, which contains a matrix of *fields*. Each field holds at most one *piece*. Whether white or black is to make a move (in two-person games) is considered part of the board state.
- There is a cursor which we call the *finger*. The finger points at one of the fields of the current board. Many statements implicitly use or modify the finger, as explained below.
- The *current direction* can be set to north, northeast, east, and so on. A ***step*** statement moves the finger one field in the current direction.
- Finally, there is a *hand*, which can temporarily hold a piece. A ***pick up*** statement removes the piece from the field currently pointed at by the finger. The piece is held in the hand while the finger can be moved across the board, until a ***put down*** statement is performed. The program in Figure 3.1 does not use the hand, but the examples given later in this section do.

The rules are Prolog-like; in fact, the compiler generates backtracking code. Since it is easier to reason about sets than to reason about backtracking, we pretend for the moment that the language is set-based. Each statement accepts a set of positions as input and applies a test or modification to each position in the input set. For each position for which the statement succeeds, the resulting positions are placed in the output set (some statements can succeed in multiple ways and yield multiple positions). Consecutive statements act as function composition.

The programmer can define functions to structure a Multigame program. Each function has a set of positions as an implicit formal argument, and implicitly returns the set of positions that results from applying the statements to the actual argument.

```
knight_move =
    find own knight,
    pick up,
    orthogonal,
    step,
    either turn 45 or turn −45,
    step,
    not points at own piece,
    put down.
```

Figure 3.2: Rules for the knight move.

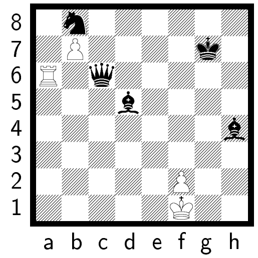


Figure 3.3: Example chess position.

Functions are allowed to be recursive. The programmer must provide a function *main*. The Multigame runtime system calls this function with a singleton input position and expects it to return the set of positions that can be reached by doing a single legal move.

In the example of Figure 3.1, the ***try*** statement tries to place a new mark onto the board. If *new_mark* fails (produces an empty set) then the game is a draw. The rule *new_mark* is defined as follows. The statement ***find empty field*** produces a set of positions; the finger associated with each position is initialized to point at a different empty field. If there is no empty field left, the result is the empty set. The next statement specifies that 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 have won the game. If the test *three_in_a_row* succeeds (i.e., is non-empty), we mark the board to be a winning node. Otherwise, the move is still legal, 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 ***find*** *own mark* statement starts from any field that is occupied by our own mark, and ***any direction*** starts looking in any of the eight directions. The ***step*** statement fails if the finger moves 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. There are many ways to implement the three-in-a-row test more efficiently; Figure 3.1 serves as a simple example.

As another example, consider the rules for a knight move in *chess*, shown in Figure 3.2. We use the chess position in Figure 3.3 to illustrate the Multigame rules. Assuming that the pieces have been declared properly (as shown in Appendix A.3), the ***find own*** *knight* statement tries to find a knight of our own color. If white is to make a move, the statement (and hence the entire function) will fail, because there are no white knights on the board. However, if black is to make a move, ***find*** succeeds in one way, with the finger initialized to point at b8.

pick up removes the knight from the board and keeps it in the hand. ***orthogonal*** continues in four different directions: ***north***, ***east***, ***south***, and ***west***. The ***step*** in northern direction fails, but the others succeed and are fed to the next statement. Note

```
bishop_move = find own bishop,
              pick up,
              diagonal,
              repeat 0..infinity times [ step, points at empty field ],
              step,
              not points at own piece,
              put down.
```

Figure 3.4: Rules for a bishop move.

```
properties
{
    passed : board : [ 0 .. 1 ]
}

main =        either normal_move or pass.

normal_move = ...
              assign (passed = 0).

pass =        try [ assert (!passed), assign (passed = 1) ] else draw.
```

Figure 3.5: Example usage of properties.

that the finger of one of the positions points at the white pawn. The ***either*** and ***turn*** statements change for each of the three input boards the direction 45 degrees clockwise and counterclockwise, resulting in six different possibilities. Three of them fail at the second ***step*** statement, because they would move off board. 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 hold in our hand. The move Nb8×a6 effectively captures the white rook at a6; the move Nb8–d7 just puts the black knight at d7.

Similarly, we can define a bishop move (see Figure 3.4). First we search for our own bishops. In the example above, we 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 northwest, northeast, southwest, and southeast, yielding 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 finally step off the board (or point at a non-empty field). We must do one additional step (because the bishop must move), and then we make sure that we are not capturing a piece of our own color. Note that replacing the 0 by 1 in the code and removing the 5th and 6th statement would not do what we want: this would exclude capture moves. Finally we put down the bishop that we have in our hand.

To increase the expressiveness of the language, we allow the programmer to define *properties*. These properties are needed to express rules for which additional state must be maintained, like checking a player's castling rights and validation of en-passant captures. Properties are named integer variables that belong to a board, a player, or act as temporary variables with a lifetime that spans the definition of a single move. A simple usage of a property is illustrated in Figure 3.5. Imagine a game where a player is allowed to pass, but ends in a draw when both players decide to pass. One way to handle this is to declare a board's property *passed* that can have the values 0 (false) or 1 (true). If the player decides to do a normal move, *passed* is set to false. If the player decides to pass, the ***assert*** statement tests whether the expression *!passed* evaluates to true, and fails if *!passed* is false. If *passed* is false, the ***assign*** statement sets *passed* to true. Otherwise, the game ends in a draw. The ***assert*** and ***assign*** statements accept all expressions that are accepted in C, except pointer arithmetics, cast expressions, and function calls. The C expressions are analyzed by the front-end compiler, because they may modify a board's or player's property. The front-end compiler generates code that changes the board's signature (see the next section) whenever one of these properties is altered.

3.3 The generated code

The Multigame front-end compiler generates game-specific program code from a Multigame program. The Multigame runtime system contains library functions that are commonly used by many games. Every game-playing program has a *move generator* (see also Figure 1.1), which is called by the runtime system when the search engine needs a list of legal moves from a given position. The entire move generator consists of game-specific code, and is generated by the Multigame front-end compiler. In this section, we will discuss how the generated code works.

Rules in a Multigame program are much like rules in a Prolog program. A Prolog program is executed by a depth-first search of the search space, looking for solutions. A Multigame move generator behaves the same way: it searches for legal moves (note that the search space of the move generator and the game-tree search space are different search spaces). For example, Figure 3.6 shows the search space for the statement sequence ***any direction***, ***step***. Assume that the finger of the sample position at the root is pointing at the lower right-hand side field, as indicated by the dot. The ***any direction*** statement generates eight output positions from each input position, and sets the directions associated with each finger to north, northeast, and so on. The ***step*** statement moves the fingers.

There are at least two techniques for translating the Prolog-like statements in the Multigame code to C code: a *set-based* and a *backtracking* approach. With the set-based approach, each statement applies an operation to all boards in the set. This corresponds to a breadth-first search in Figure 3.6. Since the set size is highly variable, this approach would require much board copying and is likely to be inefficient. We

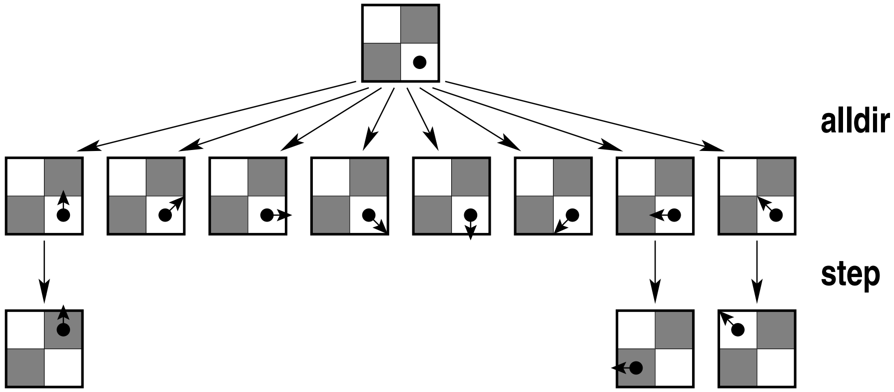


Figure 3.6: Search space for the statements ***any direction***, ***step***.

```
enum adv_retr { advancing, retreating };

typedef struct {
    board_type board;
    int        finger, dir, hand;
    stack_type stack;
} iterator_type;

extern enum adv_retr gcw_main(iterator_type *, enum adv_retr);
extern enum adv_retr gcb_main(iterator_type *, enum adv_retr);
```

Figure 3.7: The move generator interface.

therefore chose for a depth-first generation of moves. Like a Prolog interpreter, the generated code uses backtracking when searching for legal moves. Each statement is translated to two pieces of C code, one that *does* the operation (downward in the tree), and one that *undoes* the operation (upward in the tree), so that the original state is restored. These are distinguished by calling the former *advancing code*, and calling the latter *retreating code*. A stack maintains all board state that has been changed by each statement, such that the old state can be recovered quickly during backtracking.

In the remainder of this section, we will see how the move generator works. We will first discuss the interface of the move generator, then we will show how each Multigame function is translated to one or two C functions and how each Multigame statement is translated to C code.

The interface between the runtime system and the move generator is small (see Figure 3.7). The move generator exports two functions when the game is a two-player game: *gcw_main*, that is called by the runtime system when white is to make a move, and *gcb_main*, that is called when black is to make a move.[2] Compiling to two separate functions yields more efficient code. The structure for both functions is the same, and in the remainder of this section we implicitly assume that the front-end compiler emits code for white to move. A single function is exported for one-player games. Each time the function is called, it generates the next move from a given position. An alternative approach, generating all moves by a single call to the move generator, is problematic for reasons to be discussed later.

The function has two arguments: a pointer to an iterator, a region of memory that is used to maintain state between invocations of this function, and a value that indicates whether it should advance or retreat, as discussed later. The role of the function return value is also discussed later. The iterator contains the board to be expanded, the finger, the current direction, the hand, and a backtracking state stack. The move generator operates on the board in the iterator. The lifetime of the iterator spans the time to generate all moves.

The move generator uses two separate stacks. The *invocation stack* is the per-process execution stack that the C programming language offers. It holds return addresses and possibly some local variables, and grows and shrinks with the function call depth. Virtually all processors have hardware support for this stack. The *backtracking state stack* located in the iterator holds the data required for backtracking. This stack grows when executing advancing code (where data are pushed needed to restore states later during backtracking) and shrinks when executing retreating code (where the data are popped). The stack is managed in software. When *gcw_main* returns, having generated a new move, the invocation stack is empty (as far as the move generator is concerned), and the backtracking state stack contains the data to restore the state.

The Multigame front-end compiler translates a Multigame function into one or two C functions. Figure 3.8 shows how the front-end compiler translates the Multigame function *func_1* = ***any direction**, **step**, func_2*. This rule states that *func_1* first does a step in any direction, after which it calls *func_2* (provided that the step succeeds); *func_2* is another function that is defined somewhere else in the program. The code outside the shaded boxes forms the framework of the function. Within this framework, the front-end compiler first emits the advancing code for all statements within the function (the first three shaded boxes), and then the retreating code for the statements *in opposite order*. The front-end compiler emits a similar framework for all other functions it generates.

Before we show the interaction with other functions (how the function is called and how it can call other functions), we discuss how a Multigame statement is translated to C. Lines 5–8 illustrate how the advancing code for the ***any direction*** statement works. First, the current direction is pushed onto the backtracking state stack, so that it can be restored later. Then, the current direction is set to 7 (directions are numbered

[2]The *gcw_* and *gcb_* prefixes differentiate between the white and black player and avoid name conflicts with other C functions and keywords.

```
1  enum adv_retr gcw_func_1(iterator_type *it, enum adv_retr ar)
2  {
3      switch (ar) {
4          case advancing:
5              /* advancing code for "any direction" statement */
6              push(&it->stack, it->dir);
7              it->dir = 7;
8  L1:
9              /* advancing code for "step" statement */
10             if (step_table[it->finger][it->dir] < 0) goto L2;
11             push(&it->stack, it->finger);
12             it->finger = step_table[it->finger][it->dir];
13             /* advancing code for "function call" statement */
14             if (gcw_func_2(it, advancing) == retreating) goto L3;
15 L4:
16             return advancing;
17         case retreating:
18             /* retreating code for "function call" statement */
19             if (gcw_func_2(it, retreating) == advancing) goto L4;
20 L3:
21             /* retreating code for "step" statement */
22             it->finger = pop(&it->stack);
23 L2:
24             /* retreating code for "any direction" statement */
25             if (-- it->dir >= 0) goto L1;
26             it->dir = pop(&it->stack);
27             return retreating;
28     }
29 }
```

Figure 3.8: Translation for *func_1* = ***any direction***, ***step***, *func_2*.

from 0 to 7), representing the direction north-east. The retreating code (lines 24–26) iterates over the remaining directions. As long as not all directions have been processed, control is transferred back to advancing code, and subsequent statements (starting from line 9) are executed again, albeit with a different current direction. If all directions have been processed, the previous current direction is restored from the stack.

The advancing code for the ***step*** statement (lines 9–12) works as follows. First a test is done whether a step from the current position in the current direction will not step off the board (which corresponds to a failing statement). This is done by reading *step_table*, a compiler-generated table that is used to look up the new position of the finger after a step in the current direction. If the step would fall off the board, the corresponding table entry is -1, control is transferred directly to the retreating code in line 23, and the advancing code for the subsequent statements (lines 11–16) is skipped. Otherwise, the new position is read from *step_table*.

In the retreating code (lines 21–23) the finger is restored by pulling the old value from the stack.

The Multigame language contains looping statements, such as ***while*** ... ***do*** ... and ***repeat*** ... ***times*** ... . These require that both the advancing and retreating code be reentrant, so that advancing code can be executed multiple times before the corresponding retreating code has finished. Reentrancy is achieved by careful stack manipulation and by avoiding global or static data. This policy also makes the move generator thread-safe without additional effort, and this is an advantage on multi-processor systems.

By now we can explain how functions interact. In the example above, the Multigame function *func_1* calls *func_2*. Recall that all generated functions, including *gcw_func_2*, contain both advancing and retreating code. The function *gcw_func_1* calls *gcw_func_2* at two places: once in the advancing code (line 14) and once in the retreating code (line 19). To implement the semantics of a Multigame function call, *gcw_func_2* executes advancing code when *gcw_func_1* was executing advancing code, and *gcw_func_2* executes retreating code if *gcw_func_1* was executing retreating code. This is achieved with the *ar* argument, which is set by the caller. The actual argument is always a constant known at compile time, and it may be tempting to translate the advancing code and retreating code to two different functions, to save runtime overhead of passing and processing the *ar* argument. However, many statements are gotos from advancing to retreating code or vice versa. Since C does not support non-local gotos, it is not possible to generate advancing and retreating code into separate functions.

Both the advancing code and the retreating code end with a return statement. For the caller, it is important to know whether the callee ended performing advancing code or retreating code. A return from advancing code is indicated by an *advancing* return value (line 16) and implies that it was successful generating a new successor state. A return from retreating code is indicated by a *retreating* return value (line 27) and implies that no (additional) successor states could be generated and that the original state is restored. *gcw_func_1* must resume advancing code if *gcw_func_2* ended executing advancing code and *gcw_func_1* must resume retreating code if *gcw_func_2* ended executing retreating code. This is important, because *func_2* can produce any number of results, for example, if it contains a ***find own*** *pawn* statement. Assume that there is no pawn at all and that *func_2* thus will fail. Although *gcw_func_2* is called from advancing code (with the *ar* argument equaling *advancing*), *gcw_func_2* will return the *retreating* value, indicating that it was not successful in finding a successor state. *gcw_func_1* must resume execution in retreating code; therefore the conditional jump

```
if (gcw_main(&iterator, advancing) == retreating)
    error("Position has no successors\n");
else
    do
        add_next_child(parent, &iterator->board);
    while (gcw_main(&iterator, retreating) == advancing);
```

Figure 3.9: Invocation of the move generator.

from line 14 to *L3* is made. If, in contrast, there is at least one pawn then *func_2* will succeed. *gcw_func_2* will return *advancing* and the conditional jump at line 14 is not made: execution resumes in advancing code (line 15). At a later moment, the retreating code for *gcw_func_1* will be executed, starting at line 17. *gcw_func_2* will be called with the *retreating* argument, to see whether it can generate more successor states, or to undo the changes it made to the current board state. If it is unable to generate more successor states (when there is exactly one pawn), it will return *retreating* and the jump from line 21 to *L4* is not made. However, if *gcw_func_2* is able to generate more successor states (when there are at least two pawns), it will return *advancing* and control is transferred back to advancing code (line 15).

Figure 3.9 shows how the Multigame runtime system invokes the move generator to expand a node. The function *gcw_main* is generated by the front-end compiler for the Multigame function *main*. The first invocation yields the first child. Note that the Multigame program is erroneous if no child is created; a terminal position (win, draw, or loss) is expressed as a legal move and thus creates a child. The call to *add_next_child* registers the newly created child. Subsequent children are created by invocation of *gcw_main* until it returns *retreating*, indicating that no more children are available.

The function call mechanism described above is intricate. This is partly caused by the choice of the target language C, which has two severe restrictions. C does not allow non-local gotos, and does not consider labels as first-class objects (we cannot assign text segment addresses to variables and pass them to other functions). With these features, code generation would have been easier and slightly more efficient. The first restriction (no non-local gotos) forces us to generate advancing and retreating code into a single function, because the generated code contains many conditional jumps from advancing code to retreating code and vice versa. The second restriction (no first-class program code addresses) almost forces us to unwind the invocation stack for each generated successor state. Would we have been allowed to pass the address of a statement to a called function, we could have used continuation pairs to tell where it should resume execution after function return. One continuation then points at advancing code and is used when the callee succeeds; the other continuation points at retreating code and is used when the callee fails. C disallows passing addresses of arbitrary statements, but does allow taking the address of a function. By splitting a Multigame function into multiple C functions it is possible to avoid unwinding the invocation stack and use a pure depth-first approach. This would enable the merging of the backtracking state stack with the invocation stack. However, splitting a Multigame function to multiple C functions implies that extra administration is needed to find the right continuation pairs and is on average probably just as (in)efficient as the scheme we currently use.

The programmer is not obliged to use the front-end compiler to create a move generator. Instead of writing a Multigame program, it is possible to use a hand-written move generator in C. However, the interface described above is designed for use with a backtracking move generator. A programmer writing a customized move generator typically does not want to write a backtracking move generator that is called multiple times to create all possible positions. Therefore, the Multigame runtime system offers an alternative interface. In this case, the programmer must provide a function that is called to generate *all* positions. Each time a new child is ready, a runtime system function is called back to register the newly created child. In our experience, the alternative interface is considerably easier to use for human programmers.

Normally, a move generator computes the signature (see Section 2.1) for a newly created position incrementally: each time a piece is picked up or put down, the signature is bitwise xor'ed (exclusive or) with the corresponding table value. This may be expensive for games where a move changes many pieces of the current position, such as in Rubik's cube. A compile-time option exists to recompute the signature from scratch after each move, instead of maintaining the signature during a move. For Rubik's cube, it makes the move generator 37% faster and the entire application 12% faster.

We also experimented with a version of the front-end compiler that generates IA-32 assembly instead of C, albeit for a subset of the Multigame language (all statements except those that require a C expression as argument). Since the assembly language allows inter-procedural jumps, the advancing and retreating codes are generated to separate functions. A newly created position is reported to the runtime system using the alternative interface that is also used for hand-written move generators, which does not enforce unwinding the invocation stack. Continuation pairs determine what a function should do if it terminates: one continuation is used for the advancing code and the other for the retreating code. The invocation stack and the backtracking state stack are combined into a single stack, which is manipulated by the efficient ***push*** and ***pop*** IA-32 instructions. Each function has its own stack frame, which contains the continuation pairs and possibly some local variables. A frame pointer points to the stack frame of the currently executing function, irrespective of whether it executes advancing or retreating code. Since advancing code leaves state on the stack that is not unwound until the corresponding retreating code has finished, and since the frame pointer varies with the invocation depth, the frame pointer does not always point to the topmost frame. The most frequently accessed internal variables (the *finger*, *current direction*, and *signature*) are kept in registers. As shown in the next section, this experimental front-end compiler generates more efficient code. Since this experimental front-end compiler fails to support a considerable part of the Multigame language, we do not use it for the experiments in Chapter 4 and 5.

3.4 Performance of the generated code

Since a Multigame move generator is generated from a high-level description, some loss of performance with respect to a hand-coded C implementation is to be expected. One of the causes of possible performance loss is that the Multigame front-end compiler does a straightforward translation from a Multigame program to C code; such translations may be suboptimal. Just like programs written in a high-level language like C generally run slower than programs written in assembly, programs written in Multigame will run slower than programs written in C. If the performance loss introduced by the front-end compiler is unacceptable, the programmer may consider writing the entire move generator in C.

Another reason for possible performance loss is the fact that it is hard to implement incremental evaluation functions [108] in Multigame. For some games, high performance implementations incorporate part of the evaluation function with the move generator, and reevaluate only those parts in which a moving piece is involved (e.g., Deep Blue [57]). Incorporating part of the evaluation function in the move generator can be more efficient, but we do not consider this as the practice of good software engineering. Multigame does not allow mixing code for the evaluation function with the generated code of the move generator. The only way to implement an incremental move generator is to let the evaluation function itself find out the differences of the parent position and the current position. We implemented such an incremental evaluation function for the 15-puzzle. It turned out to be slower than evaluating each position completely. Other games (e.g., chess) are likely to profit from incremental evaluation, but we did not implement this. All games evaluate positions completely.

A third reason for possible performance loss is related to move ordering. A Multigame move generator generates a list of moves, after which they are sorted using various heuristics. If the move generator were able to generate a *partial*, *sorted* list of moves, the search engine could ask the move generator to generate the most promising move, and then search it. If a cutoff occurs, there is no need to generate the remaining moves. Since a Multigame move generator generates moves one at a time, it could generate a partial list, but the moves appear in random order. The Multigame front-end compiler does not have the game-specific knowledge to create a move generator that generates the most promising move first. Major changes to the Multigame language and compiler are required to achieve support for partial, sorted move generation.

The Multigame front-end compiler does some optimizations; however, we rely on the C compiler to do additional optimizations. The front-end compiler often generates improved code in the following situations:

- when it knows which field the finger is pointing at (for example, after a ***move***

to *(3,4)* statement);

- when it knows which piece is on the field pointed at (for example, after a ***points at opponent's*** *king* statement);
- when it knows which piece is currently held in the hand (for example, after ***find own*** *king*, ***pick up***); or
- when it knows the current direction (for example, after a ***north*** statement).

In these cases, it is not necessary to push state onto the backtracking state stack, because the front-end compiler can generate state-restoring code in retreating code using constants known at compile time.

A second optimization is to inline multiple Multigame functions to one generated C function. Multigame functions that are called from multiple places (including recursive functions) are not inlined.

Another optimization is done for arguments of the ***test*** and ***not*** statements. These arguments do not need to generate all possible results, but are aborted as soon as one result is found. For example, no test for a rook attack is done in the statement ***test*** *either attacked_by_knight* ***or*** *attacked_by_rook* if the current position is attacked by a knight (in this case, it does not even search for multiple attacking knights). For efficiency reasons we emit code twice for all user-defined functions: one for (direct and indirect) invocations from ***test*** and ***not*** statements that finish as soon as one solution is found, and one for normal invocation.

Yet another optimization concerns the stack operations described in Section 3.3. Since many statements modify the stack, we keep the stack pointer of the backtracking state stack in a fixed register, using the ***asm*** statements recognized by *gcc*. On the Pentium Pro, it makes the move generator approximately 15% faster. The problem with the Intel IA-32 architecture is that there are few general purpose registers, and assigning one of those registers for the stack pointer leaves even fewer registers for other computations. On architectures with many registers, it will be useful to also hold the *finger*, *current direction*, *signature* (requires two registers on a 32-bit architecture), and possibly the *hand* in fixed registers, instead of in the iterator.

A possible optimization that has not been implemented is instruction reordering for Multigame statements: sometimes it is useful to interchange independent statements to obtain strength reduction, an optimization that the C compiler will not detect and exploit. One example where strength reduction would help is the statement sequence ***any direction***, ***pick up***. The Multigame front-end compiler emits code that picks up the current piece eight times (for each possible direction), while it would be more efficient to first pick up the current piece and then start the iteration over all directions. We did not implement this optimization, because it is easy for the programmer to provide the most efficient instruction order.

We implemented the *15-puzzle*, *Rubik's cube*, and *chess* using both a Multigame program and a move generator written in C. The C implementation for chess is ported

	move generator (μs)			normalized application runtimes		
	MG $\rightarrow$ C $\rightarrow$ IA-32	MG $\rightarrow$ IA-32	C $\rightarrow$ IA-32	MG $\rightarrow$ C $\rightarrow$ IA-32	MG $\rightarrow$ IA-32	C $\rightarrow$ IA-32
15-puzzle	6.06	3.68	3.66	1.00	0.855	0.855
Rubik's cube	69.3		58.9	1.00		0.926
chess	422		44.0	1.00		0.196
double-blank puzzle	10.8	6.41		1.00	0.863	
checkers	19.1	15.7		1.00	0.977	

Table 3.1: Move-generator performance.

from CilkChess, the successor of $\star$Socrates [62]. Table 3.1 gives performance numbers for the move generators of these games. The second column gives average times for move generation using the Multigame move generator; the fourth column those using the hand-written C implementation. The last three columns show the difference at the application level. Here normalized average runtimes for a number of test positions are given; the norm is the execution time for the version with the front-end generated move generator.

The hand-written move generators for the 15-puzzle and Rubik's cube are 61% and 17.6% faster than those generated by the front-end compiler. On application level, the hand-written versions are 17% and 8% faster. For chess, the difference in performance is large, a factor 9.6 at the move generation level and a factor 5 at the application level. The main reason that the Multigame move generator for chess performs so badly, is the expensive test to avoid moves that leave the own king in check (see Appendix A.3). However, the C implementation for the chess move generator is roughly 2,700 lines of code (not including the generated auxiliary tables, but including the code to generate the tables), taking two weeks to implement (we ported the algorithms used by CilkChess, but rewrote every line of code, since the data structures used in Multigame are different). In contrast, the Multigame implementation is 208 lines, which an experienced Multigame programmer can write in a few hours.

The experimental front-end compiler that emits IA-32 assembly instead of C is able to generate code for the 15-puzzle, the double-blank puzzle (see Section 4.7.5.5) and checkers; Rubik's cube and chess contain statements that cannot be translated. The performance numbers for the experimental front-end compiler (in the third column of Table 3.1) show that the move generator for the 15-puzzle is just as fast as the hand-written C code; with some more compiler optimizations it would have generated even faster code. Also, the move generator would have profited significantly from the MMX extensions found in the Pentium III processors; unfortunately these extensions have not been implemented in the older Pentium Pro processors that we use for our measurements. The move generators for the double-blank puzzle and checkers are also faster with the experimental compiler. It is unfortunate that the experimental front-end compiler cannot translate the Multigame chess program, but extending the

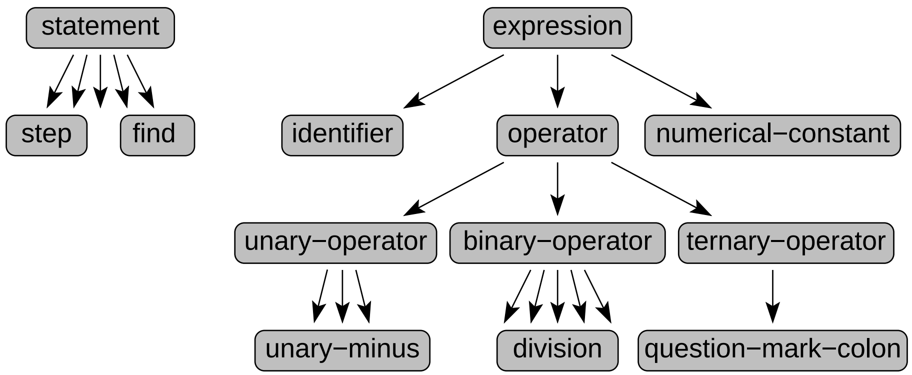


Figure 3.10: Inheritance trees in the Multigame compiler.

experimental front-end compiler to translate the entire Multigame language requires significant effort. It would have given an answer to the question on how much performance one loses by using a high-level language. For one instance, the 15-puzzle, we showed that the performance is approximately the same, but chess would have been a more interesting test case.

3.5 Experiences with an object-oriented compiler

Although the code generated by the Multigame compiler is in C, the compiler itself is written in C++ [43] (except for the lexical analyzer, which is written in Lex [1, 75], and the parser, which is written in LLgen [55]). The compiler uses the object-oriented features of the C++ language. To structure the code, the compiler makes frequent use of the ability to define *classes* and *subclasses*. There are several abstract classes[3] at the root of inheritance trees, as illustrated by Figure 3.10.

One of the abstract classes is the class *statement*. Each of the Multigame statements (e.g. ***step*** or ***find***) is represented by a subclass of the class *statement*. A sequence of statements (delimited by ***[*** and ***]*** in a Multigame program; or the list of statements associated with each user-defined function) also inherits from *statement*. When the parser in the Multigame compiler reads a Multigame program, it builds a parse tree, and attaches objects of subclasses of *statement* to the parse tree. It uses the parse tree for semantic analysis and for code generation. The two most interesting methods of class *statement* are *advancing_code()* and *retreating_code()* (see Figure 3.11). Invocation of these methods generates, respectively, the advancing code

[3] An *abstract* class is a class which cannot be instantiated, because it has at least one method declared but not defined. A subclass of this class can define these abstract methods. If a subclass defines all abstract methods derived from its superclasses, it can be instantiated.

```
class statement
{
   public:
      void advancing_code() = 0;
      void retreating_code() = 0;
   ...
};
```

Figure 3.11: ***class*** *statement.*

```
class expression
{
   public:
      void lvalue() = 0;
      void rvalue() = 0;
      void no_value() = 0;
      void retreating_code() = 0;
   ...
};
```

Figure 3.12: ***class*** *expression.*

and the retreating code for the statement. Each of the subclasses of *statement* defines these methods in a way that is specific for each statement.

Another abstract class is the class *expression* (also shown in Figure 3.10). Expressions in the Multigame language are almost as extensive as expressions in C. Many classes inherit from *expression*, such as *numerical_constant*, *identifier* (for variables), and *operator*. An *operator* is also an *expression*, and has a number of *expression*s as arguments, depending on the arity of the expression. Interesting methods for *expression* are shown in Figure 3.12. *lvalue()*, *rvalue()*, and *no_value()* generate advancing code for a expression or subexpression. Which of them is used for a particular subexpression depends on the context within the entire expression. *lvalue()* generates the code for an expression used as a lvalue (an assignable value on the left-hand side of an assignment operator); trying to use a constant or operator results in an error message from the compiler. *rvalue()* generates the code for an expression used as a rvalue (on the right-hand side of an assignment operator). *no_value()* generates the code for an expression that must be evaluated because of side effects (such as assignments), but for which the resulting value is not used (for example, the entire expression, or the left-hand side of a comma operator). *retreating_code()* generates the retreating code to undo all side effects during backtracking.

The above classes mirror syntactic elements from Multigame programs. There are a few more class hierarchies in the Multigame compiler, such as the hierarchy to describe the pieces on a field on the board, but these are not discussed here.

The use of classes and inheritance structures the Multigame compiler in a natural way. It almost forces an elegant implementation and yields a compiler that is easily maintainable. However, we feel that a compiler implemented in C++ is twice or three times as large as a compiler implemented in C. The elaborate inheritance trees and the verbosity of the C++ language to declare and define methods give rise to the increase in source code size; the complexity certainly does not increase.

3.6 Evaluation functions

High quality computer game playing requires the presence of an evaluation function with game-dependent heuristic information. Ideally, the Multigame front-end compiler should generate an evaluation function from the description of the rules of a game, but deriving a competitive evaluation function from these rules is far beyond the current state-of-the-art in Artificial Intelligence (Pell did some work on this; see Section 2.3 on Metagame). Therefore, the programmer has to provide an evaluation function. This task could be simplified by introducing an additional high-level language in which evaluation functions can be expressed easily. The programmer would then need to provide a program that describes the rules of a game and a program that judges between good and bad positions. We did not investigate the idea of a language for evaluation functions.

The interface between the Multigame runtime system and the evaluation function is simple. The programmer must provide a function that accepts a position and returns an integer. For one-person games using the IDA* search algorithm, this integer must be a lower bound of the distance to the target. A good evaluation function yields a lower bound that is close to the actual distance; the better the evaluation function, the smaller the search tree that gets built. For two-person games, the integer should represent the estimated merit for the given position; a high value indicates that the position is advantageous for the white player. Inaccurate evaluation values lead to weak play and might lead to increased search effort.

For two-player games, the programmer may optionally provide an extra function that decides if a quiet move is made (there is little change in evaluation value between the parent and child); the function requires the parent and the child position as arguments. For unquiet moves (such as capture moves) the evaluation value is less reliable and the Multigame runtime system may decide to extend the search with an additional ply. Since this decision is based on game-specific properties, we considered this additional function to be part of the evaluation.

For some games we ported evaluation functions from competitive programs. The evaluation function of our chess implementation is ported from CilkChess (a program developed at the Laboratory of Computer Science, MIT, USA), which won the Dutch Open Computer Chess Championship in November, 1996. Our checkers implementation uses an evaluation function that is ported from Chinook [104, 106] (a program developed at the University of Alberta, Canada), the current man-machine world champion. The Othello evaluation function is ported from Aïda (a program developed at the University of Leiden, the Netherlands). The 15-puzzle and 24-puzzle have state-of-the-art evaluation functions, which combine the Manhattan distance, linear-conflict heuristic [58], last-move heuristic [69], and corner-conflict heuristic [69] (see also Section 4.5.5). The double-blank puzzle is the 15-puzzle with the tile labeled “15” removed. It is a game with many transpositions, which we used for benchmarking. It uses the same evaluation function as the 15-puzzle, adapted for two blanks. The

game	number of players	move generator Multigame	move generator C	evaluation function C
15-puzzle	1	39	216	417
24-puzzle	1	50	271	482
awari	2	–	416	31
checkers	2	71	–	2,440
chess	2	208	2,683	1,790
Connect-4	2	22	–	–
domineering	2	15	–	–
double-blank puzzle	1	40	–	421
eclipse	2	61	–	26
go-moku	2	21	–	–
halma	2	34	–	50
hex	2	26	–	–
hexagon	2	69	–	–
life	0	41	–	–
lines of action	2	50	–	–
nine men’s morris	2	66	–	–
Othello	2	45	–	538
pegged	1	35	–	31
ps-Addle	2	46	–	61
quatro	2	89	–	–
qubic	2	59	–	–
Rubik’s cube	1	462	267	266
Sokoban	1	53	–	414
tic-tac-toe	2	21	–	–
towers of Hanoi	1	51	–	–

Table 3.2: Games implemented in Multigame. The last three columns show the number of lines of source code.

Rubik’s Cube evaluation is done using pattern databases [68] (see Section 4.5.4).

3.7 Discussion and conclusions

Table 3.2 lists the games we implemented in Multigame. Each game has either a move generator generated by the Multigame front-end compiler, a hand-coded move generated written in C, or both. The numbers in the right three columns indicate the number of lines of code. This excludes generated auxiliary tables, but includes the programs to generate the tables. The third column shows that most Multigame programs are small, yet give a precise definition of the rules of a game. An exception is the Multigame program code for Rubik’s cube, which is larger than the C implementation (462 vs. 267 lines). The length is the result of the verbose way each possible cube turn is expressed. In spite of the program size, the Multigame program has a regular structure and is easy to understand. The Multigame program for the 15-puzzle is also easy to understand, and is listed in Appendix A.1.

A few games that satisfy the constraints listed in Section 3.1 are nevertheless hard to program in the Multigame language, in particular Awari and Dutch draughts. The former game has the problem that a pit can be occupied by up to 48 stones. In a Multigame program, at most one piece can occupy a field. It is possible to declare 48 different pieces to represent a corresponding number of stones, but the 48 slightly different move-rules for each of the pieces would make it an elaborate program. The front-end compiler will not be able to generate an efficient move generator for this game without compiler optimizations that are hard to implement.

The problem with Dutch draughts is caused by the rule that a capture move is only valid if it is (among) the longest capture sequences. The Multigame language does not provide a way to first generate a list of candidate capture moves and later cancel all capture moves that are too short; the language's concept of variables is too simple-minded for this. To express the rule in the current language, one would have to check for each candidate capture move if there is no capture move with a longer sequence, solving an $O(n)$ problem in $O(n^2)$ time.

Another issue is the 50-move rule in chess: no more than 50 successive moves may be made without conversion (i.e., a move than cannot be undone, such as a capture or a pawn move) or a draw follows. It is easy to express this in the language, by adding a board property that counts how many non-conversions are made. However, doing so would make the transposition table significantly less useful (see Section 4.4 for a discussion about transposition tables). A transposition involving both a conversion and a non-conversion, such as the chess openings e4–Nf6–Nc3 and Nc3–Nf6–e4, would not be recognized as such, because the former sequence made two non-conversion moves since the last conversion, and the latter sequence zero. The Multigame runtime system would consider them as different positions. In fact, they are different, but one sees the difference only 98 plies deeper, and in practice chess trees cannot be searched to such depths. Our chess implementation therefore does not differentiate between these transpositions. Other chess-playing programs like CilkChess also ignore the difference, but scan the move list backward searching for conversions. The Multigame language needs an extension to do this efficiently, and therefore we currently do not check the 50-move rule.

The Multigame language is based on a combination of the Logo and Prolog programming paradigms. Other programming paradigms may suit this purpose just as well. We consider two other paradigms that can be used as a base for a language to describe legal moves.

Regular expressions are usable as a paradigm for a language to describe legal moves. Regular expressions are used to recognize character strings. Each piece used in a game can be represented by a character, the set of pieces constitute an alphabet, and each board position forms a word. A complication is that character strings are one-dimensional, and boards are two-dimensional. One way to circumvent this, is to give each regular expression a *set of directions* in which the expression applies. As an example, a bishop move or capture in chess can be expressed as follows. Assume ***B*** means "own bishop", _ means "empty field", ***!*** is a shorthand for "an opponent's piece", / means "substitute by", and the *, +, and ***[]*** pair have their usual meanings (meaning "0 or more", "1 or more", and "choose one of these" respectively). Then the expression

$$\{\texttt{diagonal}\}\texttt{B}\_*[\_!]/\_+\texttt{B} \tag{3.1}$$

would describe the move. If the subexpression before the / matches the input, the input will be replaced by the subexpression after the / (of course, the compiler must be able to bind the length of the second subexpression to the length of the first one). In words, Equation 3.1 states the following: in any diagonal direction, a bishop of our own color, followed by 0 or more empty fields, followed by an empty field or an opponent's piece can be replaced by a string of the same length, consisting of one or more empty fields, followed by a bishop of our own color. To express a sequence of checkers captures, it is useful to allow kinks (change of direction) within an expression. To ensure that the king in chess is not left in check after a move, or an intermediate field in a castling move is attacked, it is useful to allow *context sensitive regular expressions*. In the latter case, the regular expression that describes the castling fields (e.g., a rook, two empty fields, and a king) only matches the input if the context (one or more regular expressions describing that the intermediate fields are not attacked) matches as well. It is probable that regular expressions do not have sufficient power to recognize most or all commonly used patterns; stronger methods like context sensitive grammars are likely to be needed.

A language to describe legal moves can also be based on *logical expressions*. A king move can be described as follows:

$$\begin{gathered}\exists x,y\ \exists d_x,d_y \in \{-1,0,1\} : \textsf{own king}@(x,y) \wedge \neg\ \textsf{own piece}@(x+d_x,y+d_y)\\ \Longrightarrow \\ \textsf{empty field}@(x,y) \wedge \textsf{own king}@(x+d_x,y+d_y)\end{gathered} \tag{3.2}$$

The $\Longrightarrow$ in Equation 3.7 should read as "is replaced by". The first line tries to find a king of our own color, and an adjacent field that is not occupied by a piece of our color (an extra test to check that $d_x \neq d_y$ could be added, but is not necessary in this case). The last line states which fields should be replaced by which pieces.

A comparison between languages based on the three paradigms requires more research, and raises the following questions. First, the expressiveness of the languages may be different; it is not clear which kinds of rules can be expressed in one language but not in another. Second, one language may be easier to use than another. Programs written in the language based on regular expressions are probably much more compact than those written in another language, but may be hard to read. Third, for each of the languages, a compiler must be written. The implementation of one compiler may require more effort than the implementation of another. One possibility is to compile the language based on expressions to a *lex* program, and adapt a lexical scanner generator like *flex* to generate a C program that scans boards instead of input from a file. Fourth, given the rules of a particular game, it is not clear which of the compilers will generate the fastest move generator.

We did not devise languages based on regular and logical expressions. We chose to implement the language and compiler based on the combination of the Logo and Prolog programming paradigms, because the language seemed easy to use and a compiler for this language seemed to generate reasonably efficient code. This, however, does not imply that the choice to use a language and compiler based on one of the other paradigms is worse.

The purpose of Multigame was to provide a simple programming model that hides parallelism from the programmer. With respect to the "simple programming model" part, we largely succeeded. Many games that obey the restrictions listed in Section 3.1 can be expressed easily in the Multigame language, although there are exceptions. The language has constructs to express moves, captures, and promotion; even castling and en-passant captures can be expressed correctly. The "hides parallelism" part will be discussed in Section 4, since all parallelism is exploited in the runtime system, and not in the language. Although there *is* much parallelism in the generated move generators, which is even simple to discover (e.g., all ***any direction*** statements introduce independent chains that can be executed in parallel), on distributed hardware the parallelism is too fine-grained to exploit.

The Multigame front-end compiler is moderately large (12,000 lines of C++ code). The implementation uses the object-oriented features of the C++ language to structure the compiler in a natural way. Unfortunately, it also gives rise to more verbose program code than that of an equivalent compiler implemented in C. The compiler was not hard to implement, except for the devising of a way to generate code that implements backtracking. The target language C lacks features like coroutines and non-local gotos, which would have eased the generation of backtracking code. In that respect, the choice of this target language was not an unfortunate one, but the portability and efficiency of C compensate much.

For most games the performance of the generated code falls somewhat behind that of programs that are designed to play one particular game, but not by a wide margin. The chess move generator generated by the Multigame front-end compiler is slow. In case the programmer is not satisfied with the efficiency of the generated move generator, the programmer is free to write one in C, though the C program will usually be an order of magnitude larger than the program written in Multigame.

Chapter 4

An optimized game-playing runtime system

This chapter discusses design and implementation issues of the Multigame runtime system. The runtime system is a large piece of software that provides the infrastructure for parallel game playing on both distributed memory and shared memory parallel systems. Building an efficient runtime system for a distributed memory machine is challenging, because managing all communication, synchronization, data distribution, and work balancing in a complex runtime system is difficult.

The Multigame runtime system supports many games, both one-player and two-player games. These games share some program code, but each game also has its own game-specific code. Program code shared by multiple games is found in the runtime system; game-specific code is generated by the Multigame front-end compiler (see also Figure 1.1). The runtime system is implemented in a modular way, with clean interfaces between the modules. Each game uses a subset of the game-independent modules of the runtime system. The modular structure of the runtime system is discussed in Section 4.1.

Building an optimized runtime system requires a major effort. Contrary to many other applications that spend 90% of the time in 10% of the code, most (parallel) game-playing programs tend to spend their time in many pieces of code. Each optimization results in small overall performance improvement, usually ranging from a few to some tens of percents. It also means that many optimizations are necessary to build an efficient game-playing system. Although there is still room for more improvements, many optimizations are implemented, ranging from algorithmic optimizations, network interface firmware optimizations (see Section 4.4.4), and machine specific optimizations (for the Intel IA32 and, to a lesser extent, the SPARC V9 architectures).

The Multigame runtime system is designed to be portable. It runs on a variety of platforms and thread packages, including Linux, Solaris, Pthreads [78], the Amoeba distributed operating system[1] [114], and the Panda virtual machine [18].

Two concepts are important in the Multigame runtime system: *multi-threading* and *multi-processing*. Threads run within a process and communicate via shared memory. Processes (normally) run on different processors, have separate address spaces, and communicate via message passing. The Multigame runtime system is designed so that it can run on systems that provide any combination of these abstractions:

- *single-threading, single-processing*, intended for single processor machines;
- *multi-threading, single-processing*, intended for shared memory machines;
- *single-threading, multi-processing*, intended for distributed memory systems, where each machine contains a single processor; and
- *multi-threading, multi-processing*, intended for distributed memory systems, where each machine contains multiple processors.

The single-threading variants exhibit less overhead than the multi-threading variants. The reason is that the multi-threading variants synchronize on several data structures in a fine-grained manner, either through locking, semaphores, condition variables, or indivisible compare-and-swap instructions. The single-threading variants skip all these synchronization primitives, saving a considerable amount of runtime. At the application level, a *multi-threading* variant can be up to 50% slower than *single-threading* when kernel threads are used, due to fine grained locking.

The orthogonality between threads and processes was not in the original runtime system design, but introduced later. Originally, multi-processing implied multi-threading (Amoeba [114], the distributed operating system on which the first version of Multigame was running, forced the use of multiple threads anyway). The *single-threading, multi-processing* variant was added later for efficiency reasons. This variant can only be used on systems where message receipt is completely under control of the application. If the system delivers messages at any time (for example, through interrupts), multi-threading must be used, since the message handler is conceptually a separate thread that competes for the shared, protected resources with the interrupted program. The *single-threading, multi-processing* variant polls for messages outside all critical sections, therefore there is no need to protect shared resources to guarantee exclusive access.

The rest of this chapter discusses the Multigame runtime system in more detail. In Section 4.1, we discuss the general structure of the runtime system. Section 4.2 describes the search engines used in the runtime system (except for one, which is described in Chapter 5). Section 4.3 discusses the distributed job queue, and how work stealing is done. Section 4.4 describes various implementations of our distributed transposition table, and the optimizations used to reduce the communication overhead.

[1] The Amoeba version is not actively supported anymore.

Section 4.5 focuses on the other heuristics that are used to improve search performance, such as the transposition table and the history heuristic. Section 4.6 describes the Multigame user interface, and Section 4.7 shows performance results. Section 4.8 is a discussion section in which we elaborate on our experience with programming a distributed runtime system for game-tree search. In Section 4.9, we finally draw conclusions.

4.1 Overview of the runtime system components

The Multigame runtime system is a complex piece of software: over 30,000 lines of (mostly parallel) C code. This number does not include the front-end compiler and game-specific code (move generators and evaluation functions). To keep the code maintainable, the runtime system is implemented in a modular way.

Figure 4.1 shows the important modules in a game-playing program, and their mutual dependencies. All modules except the *move generator* and the *evaluation function* are part of the runtime system: both are provided separately by the programmer. Conceptually, most modules are grouped. Heuristics, the distributed job queue, threads, and inter-processor communication constitute separate groups. There is a layering between the modules. An arrow from *A* to *B* means that module *B* is used to implement module *A*. The bidirectional arrows between the synchronization primitives mean that one can be implemented using the others; this is explained later.

On top is the *user interface*; the user uses it to control the entire program by giving commands like “read a position” and “search this tree”. The user interface is described in Section 4.6.

The *search engine* has a prominent place in the modular structure. It uses many other modules, including heuristics, the move generator, the distributed job queue, threads, messages, and the evaluation function. Search engines are described in Section 4.2.

The *distributed job queue* is used by all work-stealing search engines. Except for the search engine described in Chapter 5, all search engines use work stealing to distribute the work over the processors. Work stealing and the distributed job queue are discussed in Section 4.3.

The search engines use several *heuristics* to guide the search. *Transposition tables* check for pairs of nodes in the search graph that have multiple paths from one to another; such transpositions are usually searched only once. Moreover, transposition tables are important for move ordering; search algorithms like Alpha-Beta perform best when the most promising nodes are searched first. The *history heuristic* is another important move ordering heuristic. *Pattern databases* can be used by one-player games to obtain a lower bound on the distance to solve a problem, just as an evaluation function like the Manhattan distance is used to obtain such a lower bound. *Repetition detection* checks for repetitions in the search graph. Since it is useless to move a piece forward and the immediately backward (or move a piece around until the same

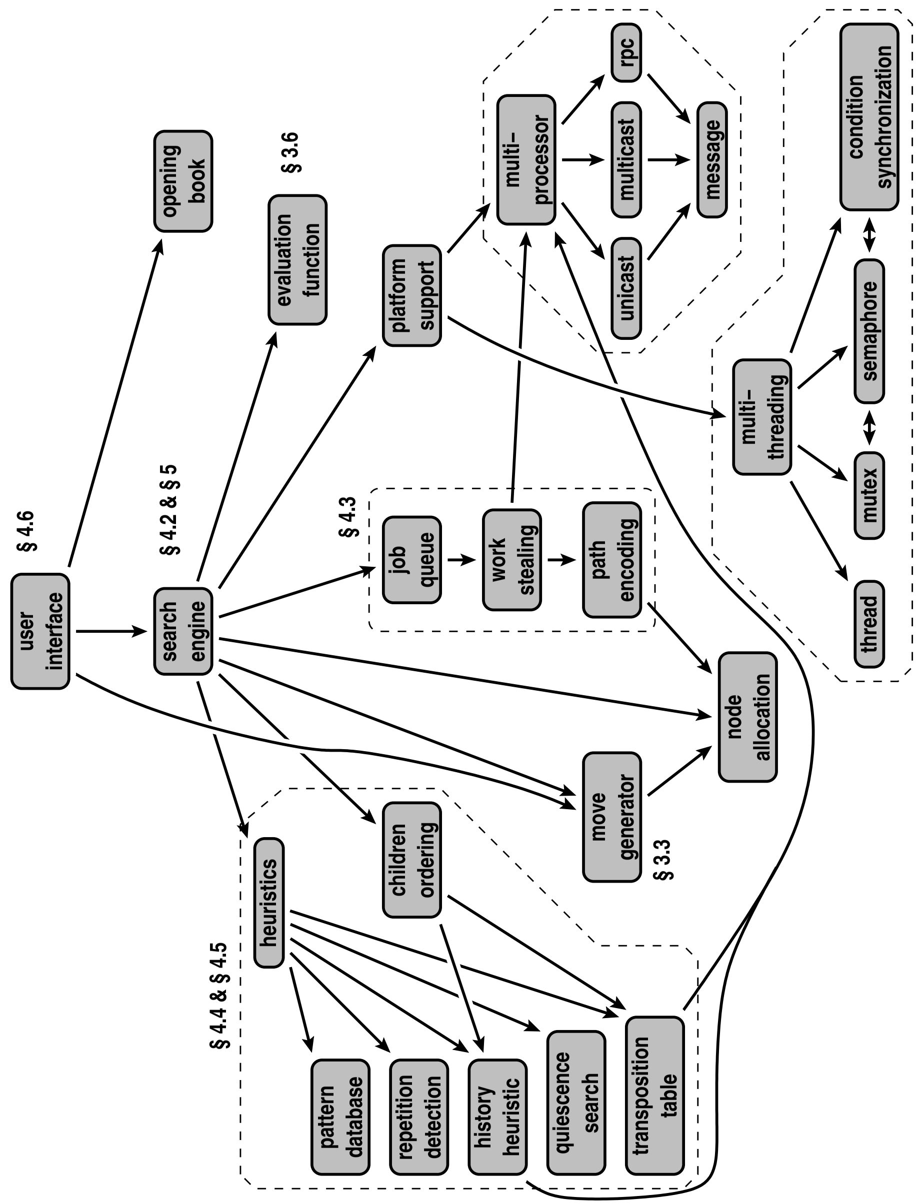


Figure 4.1: Modules in a game-playing program.

position is encountered again), repetitions are pruned during the search. *Quiescence search* is used by two-player search algorithms; this heuristic selectively extends the search at turbulent positions, for example after a capture move. It improves the quality of the search. All heuristics are discussed in Section 4.5.

The *move generator* accepts a position and returns a list of positions that can be reached by doing a legal move. The move generator was already discussed in Section 3.3.

The *node allocation* module provides the functionality to allocate and deallocate tree nodes. Its implementation is very efficient, even in a multi-threaded environment, where lock-free data structures are used to avoid as much synchronization overhead as possible.

The *multi-threading* group of modules provides an abstract interface between the thread package and the rest of the Multigame runtime system. The thread package used can be a kernel-level or user-level implementation. The interface is the same for all platforms on which Multigame runs, but the implementations differ. For example, the Panda virtual machine [18] and Pthreads [78] do not provide semaphores, but the bridging *semaphore* layer in the Multigame runtime system implements them using Panda mutexes and condition synchronization. Conversely, Amoeba [114] does not provide *condition synchronization*, but these are implemented using Amoeba mutexes and semaphores.

There is also a bridging layer for *multi-processor* support. This interface makes the communication primitives, which differ from platform to platform, uniformly accessible. Three communication primitives are supported: *unicast*, *broadcast*, and *remote procedure calls*. Earlier versions of Amoeba did not support unicast; the bridging unicast layer used Amoeba RPC to achieve point-to-point communication.

Usually, the programmer provides a game-specific *evaluation function* to let the computer play well. Evaluation functions were discussed in Section 3.6. A game-specific opening book can be provided as well. If the opening book suggests one or more good moves from the current position, one of these moves is taken randomly, instead of searching the game tree. We did not use the opening book for any of our experiments.

Not all modules are shown in Figure 4.1. There is a *statistics* module, which is used by most other modules. This module provides the functionality to let each client module collect statistics, such as the node count, or transposition table hit count. These statistics are collected per processor to avoid communication overhead. The module collects the statistics obtained by individual processors when the user enters a "statistics" command via the user interface (see Section 4.6). It has the flexibility to combine the statistics obtained by any subset of processors, and to show the statistics for any subset of modules (for example, the user can ask for the transposition-table statistics on processor 3).

In addition, the *shutdown* module is not shown. Other modules can register themselves with the shutdown module by a condition variable. The shutdown module will signal the condition variables to terminate properly upon a “quit” command from the user. Termination is done in two phases: in the first phase each processor is requested not to send messages anymore (apart from the acknowledgment that it will not do so), and in the second phase all modules clean up. This two-phase protocol ensures that no module sends a message to another processor that already terminated the receiving module.

Finally, there are several auxiliary modules implementing many simple operations on nodes, such as copying, testing for equality, reading from a file or standard input, writing to a file or standard output, and computing signatures. These functions are used by many modules.

The Multigame library is recompiled for every different game. This increases the performance of the game-playing program. If the Multigame library were an archive file or shared object against which the generated code was linked, the executable would have to evaluate expressions at run time which are now evaluated at compile time. For example, the data structure that represents a board position differs in size for different games. By recompiling the library for a particular game, the executable uses a fixed sized data structure, whose fields can be accessed efficiently. If the library were an archive or shared object, the executable would have to find out the size of this data structure at runtime, and accessing fields would have imposed much more runtime overhead. Recompiling a game takes at most a few minutes, but is often much faster.

4.2 Parallel search engines

The search engine is the heart of a game-playing program. The Multigame runtime system implements several parallel search algorithms: *IDA** for one-person games, and *Alpha-Beta*, *NegaScout*, and *MTD(f)* for two-person games. The algorithms are described in Section 2.1. In this chapter we discuss their implementations. The search engines described here are based on *work stealing* (see Section 4.3). In Chapter 5 we describe a different approach for distribution of work, and discuss its implementation for IDA*.

The Multigame runtime system also implements sequential *NegaMax* and sequential *Proof-Number Search* [2, 3]. We do not describe them here. The former algorithm is too inefficient, and was only used to generate trace files of all nodes in a tree; these trace files were used to debug the more sophisticated, parallel two-person search engines. The latter algorithm is only useful for a restricted class of search applications, and had problems with detection of transpositions [107]. These problems were solved by Breuker et al. [24]. Additional research is needed to parallelize this algorithm efficiently.

A search engine searches a tree up to a specified nominal search depth (quiescence search can extend this depth). How this is done depends on the search algorithm. For two-player algorithms, the search result returned is called the *principal variation*.

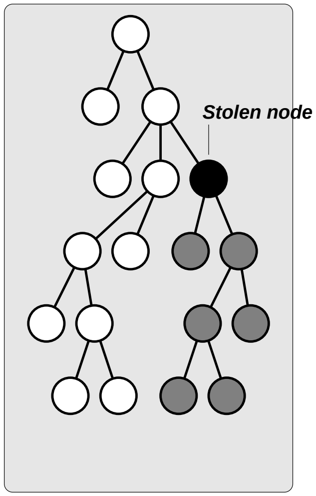


Figure 4.2: Two threads concurrently searching a tree.

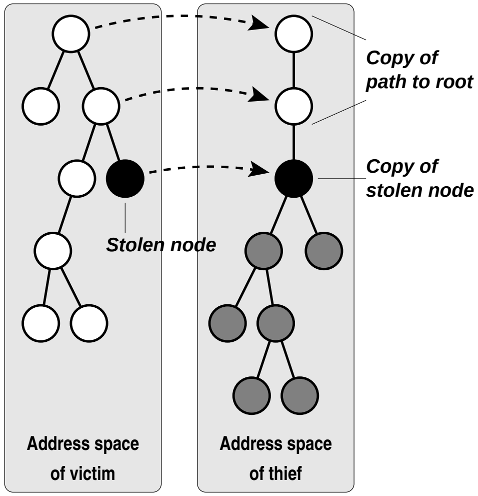


Figure 4.3: Continuation of a branch on another processor.

This is a list of nodes that represent the most probable line of play. For one-player algorithms, the search result is the entire sequence of moves that leads to the shortest solution, if such a solution is found. If the search engine can prove that no solution is possible (because all leaf nodes are lost positions or repetitions), the search engine reports the insolvability.

Each search engine is implemented as a module (see Figures 1.1 and 4.1). The programmer selects one search engine as a compile-time option. The search engines are mutually exchangeable, except that a search engine for two-person games cannot be used for one-person games, and vice versa. Exchangeability is obtained by the use of a generic interface. Each search engine implements this interface. The interface is discussed later in this section.

The search engine itself is built on top of other modules. It uses the move generator to generate the children of a node. The move generator is either written in Multigame and generated by the front-end compiler (see Section 3.3), or written in C. The search engine may also use modules that implement heuristics, such as the transposition table and the history heuristic. Furthermore, the distributed job queue, discussed in Section 4.3, is used for work stealing.

Parallelism is obtained by searching multiple subtrees in parallel. Each thread is working on its own branch, as illustrated in Figure 4.2. One thread searches the white tree nodes, while another searches the gray nodes. On a shared memory system, the nodes are allocated in the same address space. On a distributed memory system, nodes are allocated in separate address spaces, as shown in Figure 4.3. The search engines

```
RECORD NodeType IS
    Board                        : BoardType;

    Parent, FirstChild, Brother  : POINTER TO NodeType;
    NrChildren                   : INTEGER;

    RemoteProcessor              : INTEGER;
    RemoteNode                   : POINTER TO NodeType;

    IsJob                        : BOOLEAN;
    PreviousJob, NextJob         : POINTER TO NodeType;

    DepthToGo, Result            : INTEGER;
    BestChild                    : POINTER TO NodeType;

    NrChildrenLeft               : INTEGER;
    LeftBehind                   : BOOLEAN;

    Mutex                        : MutexType;
END;
```

Figure 4.4: The *NodeType* data structure.

described in this section use work stealing for distributing the work over the processors. Work stealing is discussed in depth in Section 4.3. Basically, work stealing works as follows. An idle processor, the *thief*, steals a node from a busy processor, the *victim*. The stolen node, and its path back to the root (necessary for repetition detection; see Section 4.5.3), are copied to the address space of the thief. The thief independently searches the subtree below the stolen node, and finally reports the search result back to the victim.

The *multi-processor, multi-threading* variant combines the methods depicted in Figures 4.2 and 4.3. Multiple computers distribute the nodes over the different address spaces. Within each address space, multiple threads search disjoint subtrees.

Each node in the game-tree is an instantiation of type *NodeType*, as shown in Figure 4.4. The figure shows the most important fields used by each node. The *Board* represents the current position, the side to move (for two-player games), and the signature (used for amongst others the transposition table). *Parent*, *FirstChild*, and *Brother* are used to maintain the search tree. *NrChildren* speaks for itself. *RemoteProcessor* and *RemoteNode* are used in the *multi-processor* variants; if the latter is not a null pointer, it points to the same node in the former's address space. *IsJob*, *PreviousJob*, and *NextJob* are used to link the nodes in the job queue. *DepthToGo* is a search argument, and *Result* and *BestChild* are search results; some search engines use additional fields. The use of *NrChildrenLeft* and *LeftBehind* is explained in the following paragraphs. Finally, the *multi-threading* variants use *Mutex* to serialize mutually exclusive operations on the node. Not all fields are shown. In particular, the fields required for prevention of race conditions with Alpha-Beta updates (see Sections 4.2.2.6 and 4.3.2) and FIFO ordering of messages (see Section 4.8) are omitted.

All search engines described in this section share the same structure and search subtrees by using recursion. An important design issue is the *asynchronous* nature of such a recursive call: invocation of the function with a node as argument starts the search of the subtree below it, but the function may return before the search has finished. If the recursive function returns, search of part of the subtree may still be in progress. The reason for this approach is to reduce synchronization overhead; a processor searching a node does not need to wait for the search results of all children.

The asynchronous nature of the search is illustrated with the pseudo code in Figure 4.5. The pseudo code also demonstrates the interaction between the search engine and the distributed job queue. The pseudo code is derived from the IDA* search engine, but omits or simplifies many details, which are discussed in Section 4.2.1. The figure shows two recursive procedures. The procedures do not return a value, but leave the results in the *NodeType* data structure.

The procedure *DepthFirstSearch* works as follows. First, the procedure tests whether the node is an intermediate node (line 19), i.e., not a target node and a node that does not cause a cutoff (a cutoff occurs when the evaluation value is greater than the search bound). If this is the case, it expands the node (line 20), sets the decreased search bounds of the children, and puts them into the job queue, so that they can be stolen by another processor. Then it starts trying to search the children one by one (lines 30–34). First it tries to cancel a child from the job queue. If this succeeds, the child is removed from the job queue, and the processor may search the child. If the child could not be canceled, the child was already stolen by another processor. After the loop (line 36), a check is done to see whether each of the children completed their search. This is done by maintaining a counter, *Node.NrChildrenLeft*. Each child that reports its value to the node through *UpdateParent* decrements this counter. If the counter reaches zero, all children have finished and are deleted (line 41), and the node reports its value to its own parent (line 44). However, if at least one child is still busy, the node is *left behind*. Leaving a node behind means that the processor pushes off the responsibility to update the node's value to its own parent to the thread that finishes the last child, as explained below. If the node is left behind, the procedure is immediately exited.

The procedure *UpdateParent* reports a child's search result to its parent. First, it checks to see whether the child was a job grabbed from the job queue (line 2). In a distributed environment, the child's parent may reside on another processor, and the call to *JobDone* moves the child result to the parent's processor transparently, clears *Child.IsJob*, and calls *UpdateParent* on that processor again. Then, it checks whether there is a parent at all (line 4), to see if the current iteration has finished. Next, it updates the parent's current search result by trying to lower the best result found so far. Then it decrements the parent's *NrChildrenLeft* counter (line 8). If the counter drops to zero and the parent was left behind by the thread or processor that created the

```
PROCEDURE UpdateParent(Child, Parent : POINTER TO NodeType)
    IF Child.IsJob THEN
        JobDone(Child);
    ELSIF Parent = NULL THEN
        Signal(CurrentIterationFinished);
    ELSE
        Parent.Result := MIN(Parent.Result, Child.Result + 1);
        DEC(Parent.NrChildrenLeft);

        IF Parent.NrChildrenLeft = 0 AND Parent.LeftBehind THEN
            DeleteChildren(Parent);
            UpdateParent(Parent, Parent.Parent);
        END;
    END;
END;


PROCEDURE DepthFirstSearch(Node : POINTER TO NodeType)
    IF NOT Target(Node) AND Evaluate(Node) <= Node.DepthToGo THEN
        Children           := CreateChildren(Node);
        Node.Result        := INFINITY;
        Node.LeftBehind    := FALSE;
        Node.NrChildrenLeft := Node.NrChildren;

        FOR EACH Child IN Children DO
            Child.DepthToGo := Node.DepthToGo − 1;
            JobOffer(Child);
        END;

        FOR EACH Child IN Children DO
            IF JobCancel(Child) THEN
                DepthFirstSearch(Child);
            END;
        END;

        IF Node.NrChildrenLeft > 0 THEN
            Node.LeftBehind := TRUE;
            RETURN;
        END;

        DeleteChildren(Node);
    END;

    UpdateParent(Node, Node.Parent);
END;
```

Figure 4.5: Asynchronous parallel search.

parent, the parent's parent must be updated, recursively.

The reason for this asynchronous implementation is to avoid inefficiency due to synchronization overhead. Instead of waiting until all children finished their computations, the search engine starts searching the parent's brother.

The asynchronous nature of the search engines makes their implementations intricate. A search result of a child has to be propagated asynchronously to its parent (which may reside on another machine), and if this parent update determines the final search result for the parent, the result has to be updated to the parent's parent (which may also reside on another machine). Parent updates are also implemented recursively, although the nesting depth is usually very low.

Below we discuss the generic interface of the search engines. Each search engine implements this interface. The most important variables and functions exported by the interface are the following:

- `VAR MaxSearchDepth : INTEGER;`

  This variable stores the maximum (nominal) search depth. The player can set this variable as a command via the user interface (see also Section 4.6).

- `PROCEDURE DoSearch(Node : POINTER TO NodeType);`

  This function searches the node to `max_search_depth`. Since a node structure contains a pointer to the best child, the search result (the principal variation) is stored as a linked list of nodes. Only one thread on processor 0 may call this function.

- `PROCEDURE AssistThread();`

  This is the entry point for additional threads on processor 0 and worker threads on other processors. Each thread repeatedly tries to get a job (either locally or remotely) and searches the associated subtree. Section 4.3 elaborates on work stealing.

The search engine provides several call-back functions for use by the transposition table and the distributed job queue. For the transposition table, the search engine provides a definition of the table entry type and macros that take care of the replacement of table entries, since the exact contents of a table entry and the decision strategy on replacement of table entries depend on the search algorithm. For the (distributed) job queue, the search engine provides functions that set search arguments, process search results, marshal arguments to messages, and unmarshal results from messages. These functions are also specific for the chosen search algorithm.

Below, we describe four search engines in greater detail: IDA*, Alpha-Beta, MTD(f), and NegaScout.

4.2.1 IDA*

IDA* is a well-known algorithm for one-player search (the algorithm was described in Section 2.1.4). The search algorithm is typically used for games like the *15-puzzle* and *Rubik's cube*. The search engine for IDA* is the only work-stealing search engine for one-player games implemented in the Multigame runtime system (Chapter 5 describes an alternate search engine for IDA* which is not based on work-stealing).

The pseudo code for the IDA* parallel search engine was already depicted in Figure 4.5. However, the code abstracts from many details, as listed below.

- When a processor finds a solution, it broadcasts a message to all other processors to stop searching. The pseudo code does not show the mechanism that breaks off the search. Each processor cancels outstanding jobs and cleans up its local data structures.

- The pseudo code contains race conditions and is not thread safe. The real implementation is thread safe for the *multi-threading* search engines. Each node has an associated mutex. In *UpdateParent*, the decrement of *Parent.NrChildrenLeft* and the evaluation of the condition in the next ***IF*** statement are protected by a mutex, otherwise multiple threads can draw the conclusion that they must update the parent's parent. The same mutex is used in the procedure *DepthFirstSearch*, to synchronize the assignment to *Node.LeftBehind* and the comparison of *Node.NrChildrenLeft* to zero. An execution order exists where nobody updates the parent's parent if the accesses to the data are not synchronized.

- The real search engine returns a list of moves that leads to the shortest solution, if a solution is found. The pseudo code does not show this.

- The real search engine restricts parallelism to the top of the tree. Nodes further from the root than a user-definable distance are not entered into the distributed job queue, to avoid the overhead of entering many nodes into the queue and to avoid the migration of small jobs.

- Each first child is exempt from the *JobOffer* and the consecutive *JobCancel* statements, to reduce job queue overhead. This does not restrict the parallelism; it just forces the first child to be searched by the processor that owns the parent node. Remaining children can be stolen by other processors. The subtree below the first child can be searched in parallel, subject to the restrictions mentioned in this point and the previous point.

- The search engine uses the *repetition detection* (see Section 4.5.3) module to detect multiply encountered positions, i.e., positions that also occur on their path back to the root, for example, as a consequence of forward and backward moves. This test is done in *DepthFirstSearch*, before the node is evaluated. If the test succeeds, the node is pruned and the results are reported to the node's parent.

- The search engine uses the *transposition table* (see Section 4.4) to detect transpositions. This test is done after repetition detection and before node evaluation. A transposition is pruned and the results are propagated to its parent. The transposition table is not used for move ordering as in two-player algorithms; the IDA* search engine does not order the children at all.

- In addition to the ability to use an admissible evaluation function to prune work that is too far from a target, the IDA* search engine can use a game-dependent *pattern database* (discussed in Section 4.5.4) for the same purpose. If the pattern database's value for the node's reduced position is greater than the search bound, the node is pruned and the result is propagated to the node's parent.

4.2.2 Alpha-Beta

Alpha-beta is one of the search engines in the Multigame runtime system that can be used for two-player games. The search algorithm was already explained in Section 2.1.1. In this subsection, we will describe the implementation of the parallel Alpha-Beta search engine. The MTD(f) and NegaScout search engines resemble the Alpha-Beta engine to a large extent. In the subsequent subsections, we will describe how they differ from Alpha-Beta.

The Alpha-Beta search engine searches NegaMax trees, rather than MiniMax trees. Negamax does not distinguish max-nodes and min-nodes, but maximizes the negated search results of the children at all interior nodes. MiniMax and NegaMax search exactly the same trees. The program code of NegaMax is somewhat more compact.

4.2.2.1 Child ordering

Two-player search algorithms perform best when the most promising children of a node are searched first and the least promising last. The Alpha-Beta search engine uses the transposition table and the history heuristic to order the children. Each of the children is given a priority, and the children are sorted in decreasing order of priority.

Alpha-beta uses *iterative deepening* [110]; the tree is repeatedly searched, but with increasing search bounds (one or two plies deeper). The reason for this is paradoxical, it is to search faster. Move-ordering information gathered during one iteration is used in the next iteration. This works well if the search results of the nodes visited during the shallow tree search are strongly correlated to those during the deep tree search. With a good evaluation function, this is usually the case. Due to the exponential tree growth, the last iteration takes most time. The time needed to do the shallower tree searches usually pays for the gain of time due to the good move ordering during the last iteration.

The transposition table stores amongst others the best move from a position (or the move that caused a cutoff). If a parent node is successfully looked up in the transposition table, the child that turned out to be the best move in the previous iteration gets a priority that is higher than all other children, so that it will be searched first. The remaining children are sorted according to information obtained from the history heuristic. This is explained in more detail in Section 4.5.1.

4.2.2.2 Parallel Alpha-Beta

Parallelism in Alpha-Beta is obtained by searching multiple children in parallel, much like in IDA*. The children are independently searched by different processors, and the search results are combined to determine the search result of the parent node. Work stealing (described in detail in Section 4.3) is used to keep processors busy.

The search engine of the Alpha-Beta algorithm is based on the structure of the IDA* search engine. The asynchronous approach described above is also applied to Alpha-Beta. However, the Alpha-Beta search engine is much more complicated, for several reasons explained later in this section.

There is an important difference in the structure of an IDA* game tree and an Alpha-Beta game tree. Both IDA* and Alpha-Beta build game trees that are limited in search depth, although IDA* prunes on the basis of the evaluation value, where Alpha-Beta artificially sets a maximum search depth. IDA* searches either all children of a node or none (unless a child solves the problem), where Alpha-Beta may search some of the children, pruning the remaining ones. As will become clear now, it is *the data dependency between brothers* that makes Alpha-Beta hard to parallelize.

4.2.2.3 Speculative search

The Alpha-Beta search algorithm (as well as MTD(f) and NegaScout) is inherently sequential. The search order is strict; each node in the tree is data dependent on the previous node. To obtain parallelism, it is necessary to break the data dependencies between the tree nodes, and *speculatively* search subtrees. The search bounds of a node are conservatively guessed; the search window may be wider than it would be during a sequential search. As a consequence, the parallel search engine may search nodes that a sequential search engine would have pruned.

Young Brothers Wait (discussed in Section 2.2.1) is a well-known technique to reduce the amount of redundant search effort. The search for the most promising child must have finished completely, before other children may be searched in parallel. The first child either prunes the remaining children (*if* a cutoff occurs, it is likely that the first child causes the cutoff, because it is the most promising one), or gives them a narrow search window, so that the other children are searched with reduced search effort. The Alpha-Beta search engine implements Young Brothers Wait; the next section explains how this is done.

```
  PROCEDURE UpdateParent(Child, Parent : POINTER TO NodeType)
          ...
          Parent.Value := MAX(Parent.Value, −Child.Value);
          Parent.Alpha := MAX(Parent.Alpha, −Child.Value);
5         ...
      END;
  END;


10 PROCEDURE SetSearchArguments(Child, Parent : POINTER TO NodeType)
      IF Parent = NULL THEN
          Child.Alpha     := −INFINITY;
          Child.Beta      := INFINITY;
          Child.DepthToGo := CurrentSearchDepth;
15    ELSE
          Child.Alpha     := −Parent.Beta;
          Child.Beta      := −Parent.Alpha;
          Child.DepthToGo := Parent.DepthToGo − 1;
      END;
20 END;
```

Figure 4.6: Pseudo code for Alpha-Beta search (continued on page 68).

4.2.2.4 The parallel search engine

The pseudo code for the Alpha-Beta search engine is shown in Figure 4.6. The code is simplified, and abstracts some details that are explained later.

The procedure *UpdateParent* works almost the same as in IDA* (see Figure 4.5). There are two differences. First, the function maintains the parent's search window, and narrows the window whenever a child returns a value greater than *Parent.Alpha*. Second, the resulting value of a parent is the maximum over the negated children's values.

The procedure *SetSearchArguments* sets the search arguments for a node. Initially, the root node is searched with a full window; other nodes swap and negate the parent's current alpha and beta.

The procedure *AlphaBeta* accepts two arguments: the node to be searched and a Boolean value indicating whether the node is the eldest (left-most) brother. This argument is used to implement Young Brothers Wait. Each first child is searched synchronously; the remaining children are searched asynchronously. If the *Eldest* argument equals true, the function will always finish the search of the subtree below it before returning. Otherwise, the procedure might return before the search result is known.

The procedure works as follows. First (line 22), the search arguments for the node are set. Then, for leaf nodes, the evaluation function is called (line 25), and the parent is updated (line 66). For interior nodes, the children are created (line 27), and ordered

```
PROCEDURE AlphaBeta(Node : POINTER TO NodeType; Eldest : BOOLEAN)
    SetSearchArguments(Node, Node.Parent);

    IF Terminal(Node) OR Node.DepthToGo = 0 THEN
        Node.Result := Evaluate(Node);
    ELSE
        CreateChildren(Node);
        OrderChildren(Node);
        Node.Result         := −INFINITY;
        Node.LeftBehind     := FALSE;
        Node.NrChildrenLeft := Node.NrChildren;
        AlphaBeta(Node.FirstChild, TRUE);

        IF Node.Alpha < Node.Beta THEN
            FOR EACH Child IN Node.AllChildrenExceptFirst DO
                JobOffer(Child);
            END;

            FOR EACH Child IN Node.AllChildrenExceptFirst DO
                IF JobCancel(Child) THEN
                    IF Node.Alpha < Node.Beta THEN
                        AlphaBeta(Child, FALSE);
                    ELSE
                        DEC(Node.NrChildrenLeft);
                    END;
                END;
            END;

            IF Eldest THEN
                WHILE Node.NrChildrenLeft > 0 DO
                    NewJob := JobGrab(FALSE);

                    IF NewJob <> NULL THEN
                        AlphaBeta(NewJob, FALSE);
                    END;
                END;
            ELSIF Node.NrChildrenLeft > 0 THEN
                Node.LeftBehind := TRUE;
                RETURN;
            END;
        END;

        DeleteChildren(Node);
    END;

    UpdateParent(Node, Node.Parent);
END;
```

Figure 4.6 (continued): Pseudo code for Alpha-Beta search.

with the most promising child first in the list. The first child is searched synchronously (line 32). Since it is searched synchronously, the first child's search results are updated to the current node when the (recursive) call to *AlphaBeta* finishes. A possible cutoff after the first child is detected at line 34. If there is no cutoff after the first child, the remaining children are entered into the job queue. Then, each of the children is tried to be canceled (line 40). If this does not succeed, another thread or process grabbed the job, then this child is passed over. Otherwise, if no cutoff occurred yet, the child is searched (line 42); else the child is also passed over, but not without decrementing *Node.NrChildrenLeft* (line 44).

Line 49 tests whether the current node is the eldest among the brothers. If this is the case, the search results *must* be reported to the parent before the procedure is left. Thus, as long as at least one of the children is still being searched, the processor must wait. Rather than being idle, it tries to grab another job and search it (lines 51–55). If the current node is not the eldest, asynchronous search proceeds like in IDA*. If at least one child is busy, the node is left behind and procedure is returned from (lines 58–59).

4.2.2.5 Search engine details

Figure 4.6 depicts a simplification of the real Alpha-Beta search engine. Some of the details described in Section 4.2.1 also apply to the parallel Alpha-Beta engine. The following details are omitted from the figure:

- The code shown is not thread safe. In the real code, measures are taken to synchronize multiple threads. The approach described in Section 4.2.1 is taken to prevent race conditions with respect to *Node.NrChildrenLeft*.
- A principal variation is maintained that predicts the most probable line of play. The user can ask to print this principal variation (see Section 4.6).
- Like in IDA*, parallelism is restricted to the top of the tree, to avoid the overhead of adding and removing many nodes from the job queue, and to avoid the migration of small jobs.
- As another optimization, the first child to be searched in parallel (this is the second child if Young Brothers Wait is used) is not entered into the job queue (see Figure 4.6, line 36). If it would have been entered, it would be canceled quickly anyway.
- Young Brothers Wait can be applied to the first *n* children (instead of the first one only). This can improve parallel performance for applications with wide trees and poor move ordering.
- The search engine uses *repetition detection* (see Section 4.5.3) to detect repeating positions in the path from the current position to the root. By default, the

search engine automatically assigns a draw to a node that is encountered for the third time. This conforms to the rules of many two-player games. The programmer can override this default behavior.

- The figure does not show the interaction with the *transposition table* and the *history heuristic*. The transposition table is both used for detection of positions that have already been searched, and for move ordering. A transposition table lookup is done near beginning of the *AlphaBeta* procedure. The history table is read before the children are ordered. Transposition table and history table updates occur in *UpdateParent*, when *Node.NrChildrenLeft* is decreased to zero.

- The figure shows the code for a fixed-depth search. In practice, for many games it is not desirable to search to a fixed depth. The *horizon effect* [14] makes evaluation values of some of the leaf nodes untrustworthy, especially of those for which the evaluation values are temporarily disturbed. For example, a fixed-depth search in chess would probably result in a move with a principal variation that leads to a capture move just before the search horizon, while a recapture (beyond the search horizon) may be available. Therefore the search engine uses *quiescence search* to extend the search (see Section 4.5.2). Other forms of selective search (such as *singular extensions* [5] and *null moves* [11]) can also be implemented in the search engine; however, this has not been done.

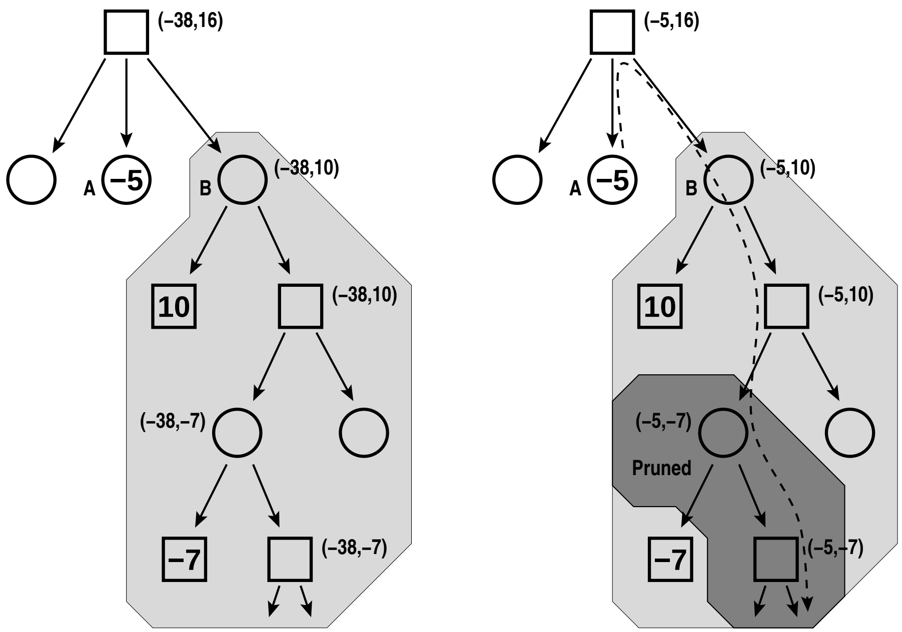
- To reduce search overhead, the current search window may be narrowed by external events at any time. This is discussed extensively below.

4.2.2.6 Window narrowing

During Alpha-Beta search, the Alpha-Beta window of a node may be narrowed whenever the result of a max-node's child is greater than alpha or the result of a min-node's child is smaller than beta. Eventually, alpha may become greater than or equal to beta, which causes the remaining children to be pruned.

During parallel search, the initial search window of a stolen node is derived from its parent's window. This window may be wider than the window used during sequential search, since the node may be stolen prior to the moment that a left-hand side's brother narrows the parent's window. This phenomenon is unfortunate, because a wider window means that less work will be pruned. It is even possible that a brother completely prunes the stolen node, in which case *all* work on the stolen node is wasted.

With speculative search windows, doing extra work is unavoidable. However, the amount of wasted work can be decreased by sending a new search window to a stolen node as soon as a window is tightened. Such an Alpha-Beta update is propagated downward in the tree, as illustrated by Figure 4.7. The initial situation is shown in Figure 4.7(a). The current (α,β) window is given between parenthesis. Node *B* was stolen by another processor, and the subtree below it (in the lightly shaded gray area)



(a) Before alpha update. (b) After alpha update.

Figure 4.7: Narrowing the Alpha-Beta window.

is being searched by this processor. The search of node *A* has just been finished, and the search result is about to be propagated to *A*'s parent.

Figure 4.7(b) shows what happens thereafter. First the window of *A*'s parent is narrowed by increasing α from -38 to -5. The new α is propagated downward, by taking the maximum of the node's current α, and the α that comes from above. It is possible that a node's α becomes greater than or equal to the node's β; in this case the subtree below it is pruned, as shown in the darkly shaded gray area.

The implementation of this downward Alpha-Beta propagation is intricate. The affected subtree may span multiple processors; each step downward in the tree might involve sending a message from one processor to another. We call such a message an *Alpha-Beta update message*. This communication is vulnerable to race conditions for various reasons. Section 4.3.2 elaborates on this.

Updating the transposition table for a node of which the original Alpha-Beta window was narrowed must be done carefully. Each table entry stores a search result tagged with a field that indicates whether this result is a lower bound, an upper bound, or exact. This tag is derived from the initial Alpha-Beta search window: if the result is

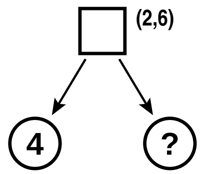


Figure 4.8: Faulty use of narrowed Alpha-Beta window.

smaller than or equal to α, the result is an upper bound; if the result is greater than or equal to β, the result is a lower bound; otherwise the result is exact. If α or β changed due to an Alpha-Beta update from above, we compare to the narrowed Alpha-Beta window instead of the initial one. Figure 4.8 illustrates a faulty scenario that might occur when comparing to the initial Alpha-Beta window. The initial search window is (2,6); the result of the first child is 4, and the result of the second child is not yet known, but assume it will be 5. Since the top node is a max-node, the result of the top node is 5. This result is *exact*, because 5 falls within the window (2,6). Now consider what happens if the window was narrowed to (2,3), prior to establishing the result of the right-hand side child. Lowering β causes a cutoff, because $4 \geq 3$, and the right-hand side child is pruned. It is incorrect to compare the best result known yet, 4, to the original search window (2,6); this would conclude that the search result is *exactly* 4. We therefore compare to the updated search window, and conclude that 4 is a *lower bound*.

Window narrowing works well: searching a chess position on 64 processors takes more than twice as much time when not using update messages.

4.2.3 MTD(f)

The Multigame runtime system also contains a search engine for the MTD(f) search algorithm, the most efficient Alpha-Beta variant currently known. The algorithm was described in Section 2.1.3. The MTD(f) algorithm differs from Alpha-Beta in two ways. First, the search window is always $(\alpha,\alpha+1)$. Second, MTD(f) searches the root multiple times (at least twice) to determine the final search result. The MTD(f) search engine reflects these differences, but for the rest it shares the same structure as the Alpha-Beta search engine, described in the previous section. Parallelism is obtained in the same way as in parallel Alpha-Beta. The implementation of the MTD(f) search engine is marginally simpler than that of the Alpha-Beta search engine.

The MTD(f) search engine uses a technique similar to Alpha-Beta window narrowing to propagate more recent search window information down in the tree. However, the search window for MTD(f) is always $(\alpha,\alpha+1)$. Narrowing the bound immediately causes a cutoff. If one child reports a value greater than α, the other children can be pruned, and if some of these children are currently being searched in parallel, this work can be killed. A killed node recursively kills its children. If a child is located on another processor, a *kill message* (rather than an Alpha-Beta update message) is sent to the node on the remote processor. The implementation of this mechanism is a little simpler than in Alpha-Beta, since work can be pruned immediately. However, the same care must be taken to avoid race conditions and to cope with non-FIFOness. No transposition table updates are done for killed nodes.

4.2.4 NegaScout

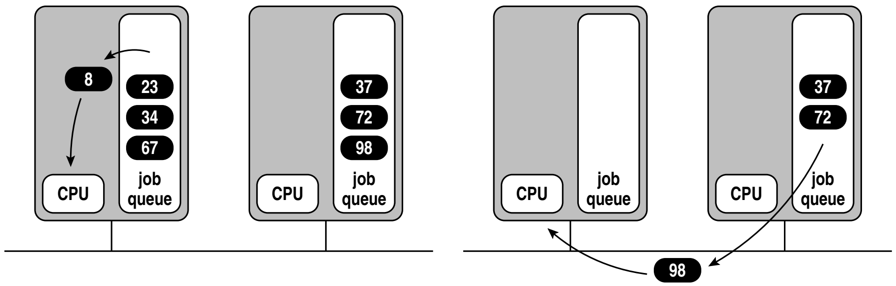
NegaScout is probably the most widely used search algorithm for two-player games (see Section 2.1.2 for a discussion of this algorithm). The Multigame runtime system contains a parallel implementation of this algorithm. The NegaScout search engine is briefly described in this subsection.

For our parallel implementation of the NegaScout search engine, we use the Jamboree search algorithm [62, 71]. Jamboree search parallelizes NegaScout by allowing the minimal window searches to be performed concurrently. Full-window re-searches are serialized: a node that needs a full window re-search must wait until all left-hand side brothers (i.e., the more promising nodes) are fully searched.

The NegaScout search algorithm is unique in the sense that it might search a node more than once (first a minimal window search, followed by a full-window search). This potentially complicates the design of the parallel NegaScout search engine. Fortunately, the implementation of parallel NegaScout can be simplified by realizing that it essentially is a combination of sequential Alpha-Beta and parallel MTD(f). Each node is either searched in parallel with a minimal window or sequentially with a full window (or both, if a re-search is necessary). Here the term “sequentially” applies only to the direct children of a node; by recursion, the children’s children may again be searched in parallel. The NegaScout search engine is implemented using the core of the parallel MTD(f) search engine, surrounded by the sequential Alpha-Beta search engine. The Alpha-Beta code runs on processor 0. Since the heart of the parallel search engine is the MTD(f) implementation, work-stealing is done for minimal-window searches only, and work can be pruned with kill messages. The Alpha-Beta code hardly forms a sequential bottleneck, because in well-ordered trees few nodes are searched with a full window (in practice, less than 0.1%).

4.3 Work stealing using distributed job queues

All of the parallel search engines described in Section 4.2 use work stealing to distribute the work over the processors. These search engines use the job queue module that transparently migrates jobs when necessary. In this section we describe the distributed job queue.



(a) Stealing a local job. (b) Stealing a remote job.

Figure 4.9: Stealing a job.

4.3.1 Local and distributed job queues

Each processor has its own local job queue. Whenever a processor needs a new job to process, it first looks in its local job queue. If the queue is non-empty, the processor grabs a job from this queue (see Figure 4.9(a)). This is a relatively fast operation that does not require communication, but, in a multi-threading configuration, does require thread synchronization. If the local queue of processor A is empty, A asks a randomly chosen processor B $(B \neq A)$ whether B has a job in its job queue. If B has a job, it returns the job to A (see Figure 4.9(b)). We call A the *thief*, and B the *victim* (although B is quite cooperative in giving away work). If B replies that it has no work, A randomly selects the next processor to ask whether it has work, until A finds a processor that returns a job.

Each local job queue is implemented as a doubly linked list (see also Figure 4.4), and maintains a simple priority scheme. When a thief steals a job from a victim, the victim will return the node that has the smallest distance to the root of the tree. If multiple such nodes exist, it returns the one inserted first, to maintain the sequential search order as close as possible. A job rooted high in the tree is likely to be larger than a job rooted low in the tree, thus this heuristic is used to migrate as few jobs as possible.

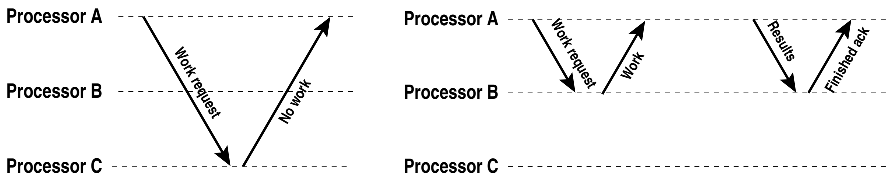
The distributed job queue has an interface that exports the following functions:

- ***PROCEDURE*** *`JobOffer(Node :`* ***POINTER TO*** *`NodeType);`*

  This procedure puts a node into the job queue. The node may now be stolen by other processors.

- ***FUNCTION*** *`JobGrab(WaitInfinitely) :`* ***POINTER TO*** *`NodeType;`*

  This function tries to grab a node from the job queue; preferably from the local



(a) Failed (b) Succeeded

Figure 4.10: Job migration protocol.

queue, but if the local queue is empty it tries to steal a job from another processor. If *WaitInfinitely* is true, it tries other processors until a victim is found. Thieves do not ask potential victims for work forever; below we describe a mechanism that prevents thieves from sending work request messages when it *knows* that a potential victim does not have work. Eventually, the function will return a job. If *WaitInfinitely* is false, the function returns a null pointer if no victim is found after a number of trials.

- ***FUNCTION*** *JobCancel(Node : NodeType) :* ***BOOLEAN****;*

  Tries to remove the node from the job queue and returns true if successfully removed; it returns false if the node was already stolen by another processor.

- ***PROCEDURE*** *JobFinished(Node :* ***POINTER TO*** *NodeType);*

  This procedure is invoked when the results of searching a node are available. If the node was stolen from another processor (rather than taken from the local queue), the results are sent back to that other processor.

Two additional functions are exported for efficiency reasons:

- ***PROCEDURE*** *JobOfferBrothers(Node :* ***POINTER TO*** *NodeType);*

  This procedure inserts a node and all its less promising brothers (children from the same parent with a lower priority) into the job queue.

- ***FUNCTION*** *JobCancelBrothers(Node :* ***POINTER TO*** *NodeType) :* ***INTEGER****;*

  Cancels a node and all its less promising brothers and removes them from the job queue. It returns the number of successful cancellations (nodes that were not stolen).

4.3.2 The job migration protocol

The job migration protocol is depicted in Figure 4.10. Figure 4.10(a) shows the messages sent for a job stealing attempt that fails. Processor *A* selects a random processor C as potential victim and sends a *work request message* to this processor. A waits for the *work reply message* from C; in this case C answers that it has no work in its local job queue.

Figure 4.10(b) shows the protocol for a successful job migration. Processor A sends a work request message to B. B removes the node from its local job queue, and transfers the work to A in the work reply message. A computes the results for the stolen node, and sends the result in the *work result message*, and starts searching for more work. B replies with a *finished acknowledgment message*, allowing A to clean up its local data structures (see below).

The work reply message and the work result message contain a sequence of nodes. The work reply message contains the path from the current root to the node that is stolen: this is necessary to detect multiply repeated board positions (see Section 4.5.3). The work result message can carry a principal variation. Internally, the distributed job queue uses a module that marshals and unmarshals sequences of nodes into and from messages. Typical work request and result messages are at most a few kilobytes large.

Before constructing the work request message, the thief preallocates a node in its own address space and sends the address (pointing in the thief's address space) in the request message to the potential victim. In case of a successful job migration, the thief will use this node to store the copy of the stolen node. The reason for sending this pointer in the work request message is to ensure that the victim knows where the thief stores its copy of the node, so that the victim can send an Alpha-Beta update (see Sections 4.2.3 and 4.2.4) for this node to the thief. In the work reply message, the victim sends the address of the original node (pointing in the victim's address space) to the thief. The thief uses this address in the work result message, to tell the victim which node contains the result.

The *finished acknowledgment message* is used to avoid a race condition and tells the thief that it can release the preallocated node. Releasing the node immediately after sending the work result message is incorrect, since the victim may send an Alpha-Beta update message or kill message before it receives the work result message. The thief would then process the update or kill message for a node that has already been deallocated. The finished acknowledgment message guarantees that the victim will not send update or kill messages for this node to the thief any longer. Of course, the update/kill messages and finished acknowledgment message must be delivered in FIFO order, otherwise the race condition still exists. Section 4.8 treats FIFO ordering and race conditions in more detail.

4.3.3 Reducing the amount of unsuccessful work requests

With the work stealing mechanism described above, we observed that during the time that there is little parallelism (primarily at global synchronization points of the search engine), most processors are idle and start sending work requests at a high frequency. The few processors that do have work to do but do not have work to distribute are assailed with work requests and hardly make progression. We therefore implemented a simple optimization that prevents a thief from sending successive work requests to the same potential victim.

Each processor maintains an array of Booleans, indexed by processor number, indicating that the indexed processor potentially has work. When a thief randomly chooses a victim, it checks whether the victim potentially has work. If this is not the case, another victim is chosen. Otherwise, the victim is asked for work. The victim either returns work, or replies that it has no work; in the latter case the thief clears the Boolean variable and will not ask the victim for work again. The victim maintains in a separate list that it has informed the thief that it has no work. Whenever the victim enters work in its empty local job queue, the victim sends a *have work message* to all thiefs that were informed that the victim had no work. In most cases, only a few thiefs must be informed and the victim sends unicast messages to these thiefs. If the number of thiefs exceeds a certain threshold, the victim broadcasts a message to all processors.

We observed that this optimization reduces the amount of communication in behalf of work stealing by up to a factor of 3. We do not claim that this is the most efficient way to reduce the amount of unsuccessful work requests, but this optimization is easy to implement and works quite well.

4.3.4 Related work

The parallel search algorithms described above are based on work stealing. This is a well-known technique to distribute the work over processors. Work-stealing based schedulers are quite commonly used by parallel game-playing programs, for example by ⋆Socrates [62] and Zugzwang [49].

The Multigame runtime system is built on top of a virtual machine that provides threads and message passing. There are, however, virtual machines with a higher abstraction level, providing implicit work-stealing and data distribution. Cilk [20, 51] offers a programming model that is particularly well suited for implementing a runtime system for parallel game-playing based on work-stealing. This is not surprising, because the development of Cilk is largely stimulated by CilkChess and its predecessor ⋆Socrates.

Cilk is a general purpose programming language (an extension to C) with explicit parallelism, and comes with a runtime system. The programming model behind Cilk uses fine-grained, independent threads. New threads can be spawned like parallel function invocations. A *sync* statement waits for all spawned children; an implicit *sync* is done at thread termination. Whenever a child thread terminates, its results can be processed by an inlet function that lives in the same scope as the parent thread, so that the parent thread's state can be modified. Spawned children can also be aborted. Memory is so called DAG-consistent [19] (this name comes from the Directed Acyclic Graph that is formed by the dependency graph of the running threads). DAG-consistency implies that a thread j sees the write to a shared object v written by thread i if i precedes j in the DAG. The hardware may be able to provide stronger consistency, but relying on this assumption makes a program non-portable. The runtime system implements work stealing: whenever a processor becomes idle, it steals the largest forked job from a randomly chosen processor. The overhead for spawning a thread is small: a few instructions in the normal case where the thread is not stolen by another processor. Cilk provides an abort mechanism to prune subtrees in NegaScout and MTD(f) search, although the example NegaScout implementation given in [113] is hard to read and understand. The abort mechanism is strong enough to simulate NegaScout and MTD(f) kill messages (see Sections 4.2.4 and 4.2.3). It is, however, not possible to narrow Alpha-Beta windows of subtrees that are being searched, as described in Section 4.2.2.6.

There are techniques other than work stealing to distribute the work over processors. The APHID search algorithm (see Section 2.2.1) statically assigns nodes at a certain depth to a slave processor and repeatedly searched the subtree below it on that processor. Unlike work stealing, the node never moves to another processor. Although this might lead to some load imbalance, the authors report that this still is a reasonable way to balance the load. The ABDADA search algorithm (see Section 2.2.1) does not use work stealing at all, since each processor starts searching the tree at the root. When a processor visits a node, the transposition table is used to see if another processor is also working on the same node.

4.4 The distributed transposition table

Game-playing programs perform best when they search the best moves first [76], and prune needless work. Many game-dependent and game-independent heuristics that guide the search in the right direction and prevent doing unnecessary work are known. The Multigame runtime system implements several game-independent heuristics. In this section, we will discuss the implementation of *transposition table*, one of the most important heuristics; in the next section we will discuss other heuristics.

In the previous sections, we discussed how the Multigame runtime system searches game trees to decide upon the best move from a given position. Actually, the name "game tree" is a misnomer, because the search space is a graph rather than a tree. Most games allow positions to be reached via different sequences of moves; these positions are called transpositions. For example, the chess openings e4–Nf6–d3 and d3–Nf6–e4 both yield the same position (see Figure 4.11).

The transposition table [23, 110, 122] is a key technique to detect and exploit transpositions. Such a table is essentially a large, possibly set-associative, cache that stores intermediate search results. Each time a board position is to be searched, the search engine checks the transposition table using a *lookup* operation to see whether the position has been searched before. If this is the case, the result of a previous search is returned only if the node was searched to at least the depth required for the current

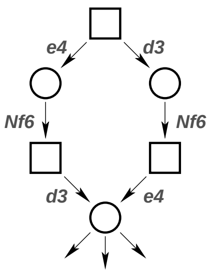


Figure 4.11:
A transposition.

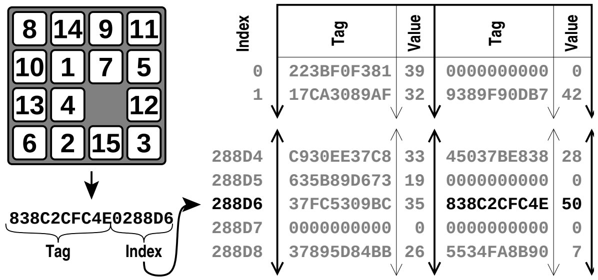


Figure 4.12: Signature mapping.

iteration. Otherwise, the table results are not accurate enough and the children of the node must still be searched. However, the table results from the shallow search are useful for move ordering: the best move from a shallow search is likely to be the best for the deeper search, and so is considered first (the remaining children are ordered by other heuristics). After the children are searched and the value for the current node is determined, the results are stored in the table using an *update* operation, possibly overwriting older information.

Transposition tables are important for performance, for three reasons. First, the transposition table prevents duplicate searches of the same subtrees. Second, the transposition table is the most important heuristic for move ordering, in combination with iterative deepening [76, 102, 110]. Third, recall that the MTD(f) algorithm needs multiple minimal-window searches to establish the root's minimax value for a tree with a certain depth (see Section 2.1.3). The transposition table is used to memorize which states have been visited during previous minimal-window searches.

The transposition table looks like a (set-associative) memory cache, but there are several important differences. First, the hit ratio of a lookup in the transposition table is lower than that of a typical instruction or data cache; depending on the size of the table and the game being played, hit-ratios of 5 to 50 percent are common. However, a hit can avoid the search of a complete subtree, thus the potential benefit of a hit is much higher, especially if the hit occurs for a node high in the tree.

Second, a transposition table tolerates a weaker coherency model than a memory cache does. A stale entry read from a memory cache obviously contains wrong data, but reading an out-of-date entry from a transposition table only results in extra search effort, not an incorrect answer.

For many games, a state representation occupies too much memory to store the entire state in the transposition table. Therefore, each board position is hashed to a 64-bit value called the *signature*. If the mapping from board position to signature is not perfect (which is usually the case) there is a small but non-zero chance that multiple positions map to the same signature. In the most extreme case, this could lead to

```
RECORD TransTableEntry IS
    Tag      : INTEGER;
    Value    : INTEGER;
    Bound    : (LB, UB, Exact);
    Depth    : INTEGER;
    BestMove : INTEGER;
    Aged     : BOOLEAN;
END;
```

Figure 4.13: Table entry structure for Alpha-Beta, MTD(f), and NegaScout.

```
RECORD TransTableEntry IS
    Tag        : INTEGER;
    LowerBound : INTEGER;
END;
```

Figure 4.14: Table entry structure for IDA*.

a non-optimal move being chosen; in practice, virtually all game-playing programs accept this risk [13, 48, 71, 104].

The Multigame runtime system computes a signature as described by Zobrist [122] (see Section 2.1). Unless stated otherwise, the Multigame front-end compiler generates a move generator that maintains the signature incrementally, i.e., the signature, changes each time when a piece is removed from or placed onto a field (the alternative is to let the runtime system recompute the signature from scratch after each move).

Indexing the transposition table is done as shown in Figure 4.12. The lower bits of the signature are used as index in the transposition table; the high bits are stored in the entry as tag and are checked each time a lookup or update is done. The table shown in Figure 4.12 is two-way associative; both entries in the indexed line are checked. If the tag does not match, the entry belongs to another node. If the tag does match, the entry most likely belongs to the same node.

The exact contents of a table entry depends on the search algorithm. The contents used by the two-player search algorithms Alpha-Beta, MTD(f), and NegaScout are shown in Figure 4.13. The *Tag* is used to distinguish positions. The field *Value* stores the search result for the corresponding node, and *Bound* indicates whether this is a lower bound, upper bound, or exact. *Depth* stores the search depth to which the node was searched. *BestMove* identifies the child that is most likely the best. If *Depth* is zero, the entry caches an evaluation value, and *BestMove* is void. The field *Aged* is used to mark the entry as being *old*. After the tree is searched completely and the computer made a move, all entries in the table are marked as old. Old entries can still be used in the next search, but if multiple positions compete for a place in the same cache line, old entries are evicted first, as described later.

The contents of the transposition table entries for IDA* are shown in Figure 4.14. Each entry also contains a *Tag* field. The field *LowerBound* stores the minimum distance from the corresponding position to a target, as the result of searching the position.

The Multigame runtime system uses two-way set-associative tables, and applies a replacement scheme described by Breuker et al. [25]. The *Aged* and *Depth* fields determine which entry will be replaced upon an update conflict. Such a conflict occurs

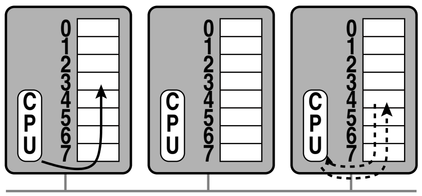


Figure 4.15: Non-shared transposition table.

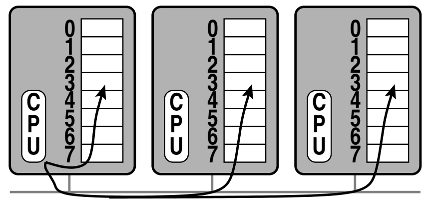


Figure 4.16: Replicated transposition table.

when both entries in a line are used and the tag of the new update does not match any of the old tags in the line. The new entry is always stored in favor of one of the old entries. The Alpha-Beta, MTD(f), and NegaScout search engines first check whether one of the old entries is aged. All entries are marked as aged after one of the players moves. If exactly one of the old entries is aged, that entry is replaced by the new one, otherwise the old entry searched to the smallest depth (considered the least valuable entry) is replaced by the new one. The IDA* search engine replaces the entry with the lowest search result (*LowerBound*).

When a tree is searched in parallel on a distributed memory multi-processor, it is often important to share the table between the processors. If the table is not shared, processors do not know from each other which nodes they searched. Transpositions may be searched multiple times by different processors, resulting in a significant search overhead.

Since transposition tables are accessed frequently (both read and written), an efficient implementation is one of the challenges of distributed search. In this section, we describe two distribution approaches for transposition tables, and one non-shared approach. To allow for a fair performance comparison, we have built highly optimized implementations for all approaches. Our optimizations are described later in this section. We will also compare the performance of the approaches for several applications.

4.4.1 Non-shared transposition tables

The simplest transposition-table variant is to let each processor maintain its own version of the table (see Figure 4.15). No lookups (dashed lines in the figure) or updates (solid lines) are communicated to other machines. A lookup merely succeeds when the same node was searched previously by the same processor. This probability decreases when the number of processors increases. It usually results in a large search overhead due to duplicate searches of transpositions by different processors. The advantage of this implementation is that there is no transposition table communication at all.

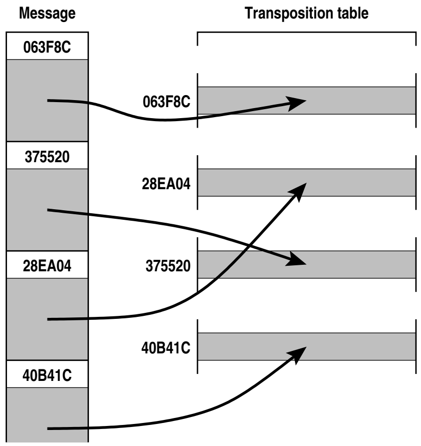


Figure 4.17: Broadcast message receipt.

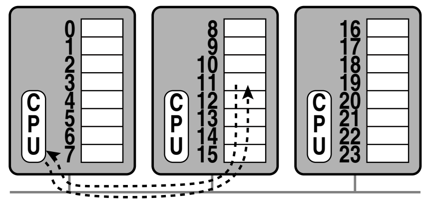


Figure 4.18: Partitioned transposition table.

4.4.2 Replicated transposition tables

One possible sharing strategy is the *replicated* implementation: each machine stores a copy of the entire table (see Figure 4.16). All lookups are performed locally and do not require communication at all. If a table entry is changed, however, the new value must be *broadcast* to all other machines, to update the replicas. The main problem with replicated tables therefore is the communication overhead required for update operations. In particular, on a large-scale system, many processors will broadcast messages to all other processors. Each processor will thus have to handle a large number of messages.

To address this problem, our implementation of *replicated* tables uses *message combining* to decrease the communication overhead. Instead of broadcasting update operations immediately, each processor stores its operations in a fixed-size buffer and broadcasts the entire buffer when it is full. This optimization greatly reduces the number of broadcast messages and the protocol overhead of handling incoming broadcasts. The best buffer size can be determined empirically.

As a result of this optimization, there is a temporal inconsistency between the replicas of the table. Recall, however, that reading stale entries does not affect correctness, it only hurts performance due to potential increase of search overhead. We therefore use the fact that the transposition table can be implemented using a weak consistency protocol.

Message handlers for broadcast messages should be as simple as possible, because a single broadcast message invokes a handler on *all* machines, imposing a substantial load on the receiving processors. We therefore do as much work as possible on the sending machine, so most work needs to be done on one machine only. The sender determines the address of the place where all receivers should store an entry. Both the address and the entry data are broadcast, so all broadcast handlers merely need to copy the data to the given address, as shown in Figure 4.17.

The greatest advantage of the *replicated* strategy is that all reads can be done locally. In contrast, updates are expensive and this replication strategy may not scale to a large number of machines, due to the quadratically increasing amount of communication traffic and the overhead of invoking handlers on all machines. In Section 4.4.6 we describe a way to trade communication for computation, to prevent machines from flooding the network.

4.4.3 Partitioned transposition tables

Another approach to share the table is to split the table in disjoint parts of equal size and store each part on a different machine (see Figure 4.18). The index number of an entry determines on which machine the entry is stored. Both reads and updates are done remotely, except when the entry happens to reside on the same machine that does the read or lookup. The read requires *synchronous* communication; the processor issuing the read waits until the reply is received. The update is done *asynchronously*, so the sending processor immediately resumes computing after the update message has been sent.

This approach has several advantages and disadvantages compared to *replicated*. The main disadvantage is that both reads and updates require communication unless the entry is stored locally. However, the chance of being stored on a remote processor is $\frac{p-1}{p}$. For a large p, almost all lookups and updates are remote, whereas *replicated* can do all lookups locally. Especially the *lookups* are expensive, since the client processor has to wait for a reply.

One advantage of the approach is that the amount of communication increases roughly linearly with the number of machines. Therefore, we expect *partitioned* to scale better on a large number of machines, provided that the aggregate bandwidth of the network grows linearly with the number of processors as well.

Another advantage of this approach is that the number of entries in the table increases linearly with the number of processors, assuming that each machine reserves a fixed amount of memory for the transposition table. Due to the increased total number of entries, fewer collisions will occur, fewer entries will be dropped, and the hit ratio will increase. Therefore, the search engine will search fewer nodes.

The message combining optimization we did for *replicated* is, to a limited extent, also applicable to *partitioned*. Unfortunately, the *lookup* operations are the ones that are expensive, and they cannot be combined due to their synchronous nature. The *update* operations are combined by maintaining a small update queue for each destination. When the update queue for a particular destination becomes full, the message is sent to the destination processor (asynchronously), where all updates are processed at once. A disadvantage of this approach is that a remote *lookup* has to check the local update queue of the destination as well, because a synchronous lookup request message can easily overtake a queued update request. To keep the check overhead for a lookup low, and to minimize the delay before an update becomes globally visible, we queue up to 4 updates per destination.

In the following sections, we discuss two novel and one well-known optimization for remote table accesses. First, we decrease the roundtrip time as much as possible by modifying the firmware running on the network interface processors. Second, we hide part of the latency by prefetching remote lookups. Third, we avoid remote accesses for which the potential gain outweighs the costs. The latter optimization also applies to replicated transposition tables, albeit for them the *updates* are performed selectively.

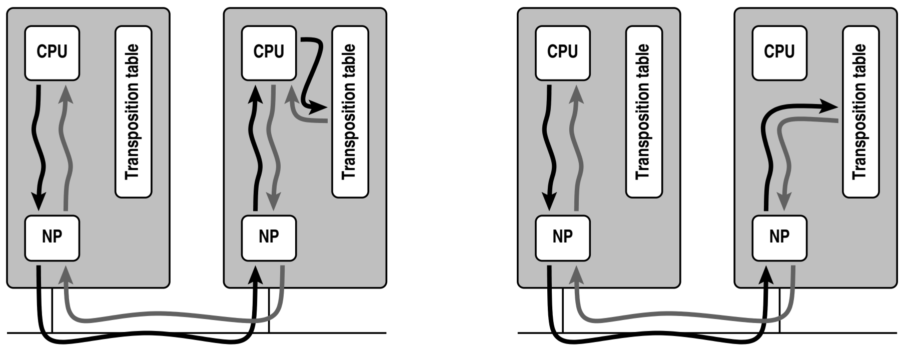
4.4.4 Customizing network firmware

Remote transposition table lookups are expensive due to their synchronous nature; the requesting processor stalls until a reply arrives. In this section we describe an optimization that decreases processor idle times by minimizing the request-reply roundtrip latency.

For communication purposes, our machines are connected by a Myrinet [21], a switched, 1.2 Gbit/s duplex network. Each network interface is equipped with a programmable 37 MHz LANai RISC processor, which can transfer data to and from host memory via the PCI bus by means of DMA. These DMA transfers are coherent with the CPU's memory caches. The network processor (NP) is slow compared to the superscalar, 200 MHz Pentium Pro CPU. Each network interface contains 1 MB of SRAM memory. The CPU can access this memory both using DMA and programmed I/O.

Remote lookups

Before we discuss the optimization that reduces the remote lookup latency, we analyze the data flow for a remote lookup in the current setup, and analyze the bottlenecks. The sequence of actions performed for a remote table lookup is depicted in Figure 4.19(a). The dark lines represent the data flow for a remote lookup request; the light lines represent the data flow for the reply. We distinguish the *client* as the processor that needs to perform a remote lookup, and the *server* as the processor that stores the desired entry. First, the client CPU assembles a *lookup request message*, which is dispatched to the client's NP. Then, the NP forwards the message over the network link to the NP on the server's network interface. The server's CPU frequently polls the server's NP to see if a new message has arrived. When the CPU sees the lookup request message, it performs the lookup in its local table, and assembles a *lookup reply message*. The reply message follows its way back via the server's NP, the client's NP, and the client's CPU. The client's CPU, which is waiting for the reply, resumes normal operation after receipt of the reply message.



(a) Lookup done by the CPU. (b) Lookup done by the NP.

Figure 4.19: Data flow for remote transposition table lookup.

The problem with this approach is the slow interaction between the server's NP and the server's CPU. The CPU is usually busy expanding and evaluating nodes, and does not continuously poll the NP to see if a message has arrived, but rather checks the NP at a regular interval. The client's CPU idles during the time that the server's CPU fails to poll its NP. The server's NP could generate an interrupt to signal the CPU that a message has arrived, but this has two disadvantages. First, the interrupt causes a kernel context switch, and the application receives a signal. The overhead for delivering an interrupt to a user-level process is approximately 31 μs, which is large compared to the polled roundtrip times of 44–53 μs. Second, the interrupt may occur when the CPU is in a critical section. The Multigame runtime system *can* handle this, provided that the runtime system is compiled with the right options (*multi-threading*, *multi-processing*) enabled. However, the *single-threading*, *multi-processing* variant, which polls only outside critical sections, is much more efficient, since it does not protect critical sections with locks. For an extensive discussion of interrupt vs. polling-based message delivery, see [74].

Low-latency remote lookups

In the remainder of this section, we describe an aggressive optimization to reduce the communication overhead. We exploit the flexibility that reprogrammable NPs offer, and customize the firmware running on the NP in such a way that the remote lookup operation on the partitioned transposition table lookup is performed by the NP itself, rather than dispatched to the CPU. Removing the server's NP–CPU interaction from the critical path of a remote lookup expectedly decreases the latency by a significant amount.

Basically, the remote lookup operation works as follows, and is illustrated by Figure 4.19(b). The CPU of the client builds a lookup request message and dispatches it to the client's NP, which forwards it to the server's NP. The server's NP reads the desired transposition-table entry without intervention of the server's CPU, and returns the table entry in a reply message to the client's NP. The client's CPU reads the reply message and continues normal operation.

Since we use two-way associative tables (see Section 4.4), the server's NP returns both entries in a line. Upon receiving the result, the client's CPU checks whether one of the tags matches the one it is interested in. Alternatively, the server's NP could check the tags and return at most one entry (depending on whether one of the tags matches). The CPU, however, performs the check much faster than the NP and since table entries are small, the cost of sending a few extra words is negligible.

All transposition table entries are stored in main memory. We do not store entries in NP memory, because this memory is too small to accommodate a reasonable amount of table entries (our network interfaces are equipped with 1 MB of SRAM memory). The NP uses its DMA engine to read table entries from main memory, because the LANai NP cannot access main memory by means of programmed I/O. Since the DMA engine of the NP uses physical addresses, the NP processor stores a copy of the part of the page table that maps the virtual addresses of the transposition-table pages to physical addresses in NP memory. The NP uses this page table to translate virtual transposition-table addresses to physical addresses. We pin the memory pages of the transposition table to prevent the operating system from swapping them out or altering their page table entries, so the NP's copy of the page table remains valid until the process terminates. Since the Pentium Pro uses 4 KB memory pages, the relevant copy of the page table of a 64 MB transposition table occupies 64 KB of NP memory (16,384 entries of 4 bytes each).

Implementation

We implemented our customized firmware as an extension to the general-purpose message-passing library LFC [17]. The LFC software runs partially on the CPU and partially on the NP. We call the implementation that uses customized network firmware *Custom*, and the implementation that uses native firmware *Native*. *Custom* extends *Native* with two message types: a transposition table lookup request and a lookup reply.

Our customized implementation requires two modifications to the Linux kernel, but the same modifications are also needed by native LFC. First, we modified the Linux kernel to allow non-root users to pin memory pages; other operating systems may (BSDI) or may not (Solaris, FreeBSD) allow this without modification. Second, a process' memory map is exposed by a kernel module that translates virtual to physical addresses on request. At program initialization, the CPU uses this module to build a copy of the part of the memory map that contains transposition-table pages into NP memory.

On the server side, there is no concurrency control between the CPU and the NP. Only the CPU can write a table entry, but both processors can read an entry. There is a small chance that the CPU writes an entry while the NP is simultaneously reading the same entry. Synchronizing the CPU and the NP is expensive, since neither the Pentium Pro, nor the LANai NP can do an indivisible read-modify-write operation (e.g., test-and-set) on each other's memory. To obtain mutual exclusion, a software solution like Peterson's locking algorithm [85] is needed, but in [16] we experienced a high overhead for such an approach, because multiple uncached memory references across the I/O bus are needed to access a lock. Therefore, we decided to allow race conditions between the CPU and the NP, making sure that reading an entry while the CPU is writing the same entry is harmless. We do this as explained below.

The size of a table entry is 8 or 12 bytes, depending on the search algorithm. The LANai NP requires a 2 or 3-cycle DMA transfer to read an entry, which might be interleaved by write cycles to the same entry by the CPU. The "tag" field in an entry (approximately 43 bits wide) is spread across the table entry such that each 4-byte word contains part of the tag. The remaining bits in each word are used for storing the rest of the entry data, such as the search result. When the CPU writes a *new* entry (with a different tag), the tag of the entry in memory temporarily matches neither the old tag nor the new one. If the NP reads the entry during that time, the requesting CPU will receive a scrambled tag and decides that the lookup will not succeed.[2] It is also possible that the CPU *updates* an entry that was already in the table, for example, after searching a position to an extended depth. In this case, the tag will not change, and if the NP reads the entry while the CPU updates it, the client CPU will use the scrambled data, provided that the entry happens to contain the data for the node that the client was interested in (the client could also be interested in data for another node that maps to the same position in the transposition table). The client might use the search result that belongs to another (deeper) search depth, or might order the children wrongly because it uses a stale "best field" entry. This cannot lead to a wrong search result.

Performance

We compared the bare performance of *Native* and *Custom* using two micro benchmarks. The first benchmark performs remote lookups to random destinations as fast as possible. Table 4.1 lists the remote lookup times and the maximum number of remote lookups per second for various numbers of processors. When more processors are added, the remote lookup times increase through network contention, and through increased network lengths. Each Myrinet network switch adds a 100 ns. latency. The minimum distance between two network interfaces in our network topology is 1 switch, the maximum distance is 10 switches. The difference in roundtrip latency between the least and the most distance network interfaces is therefore 1.8 μs.

[2] Except in the unlikely case that the tag matches the signature of an unrelated position by accident; in practice, such small risks were already accepted because the hashing scheme is imperfect as it is.

	Native		*Custom*	
CPUs	**time (μs)**	lookups (/s)	**time (μs)**	lookups (/s)
2	**44.1**	22,676	**27.0**	37,037
4	**50.3**	19,868	**29.6**	33,822
8	**51.9**	19,231	**30.6**	32,710
16	**52.5**	19,036	**31.0**	32,258
32	**52.7**	19,018	**31.2**	31,555
64	**52.7**	18,976	**31.5**	31,349

Table 4.1: Remote lookup latencies and throughputs for *Native* and *Custom*.

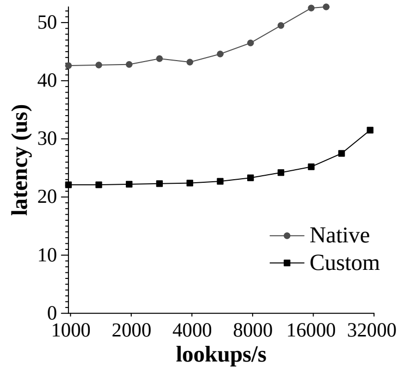


Figure 4.20: Latencies under varying contention on 64 processors.

Note that the numbers for *Native* overestimate the performance of real applications, because the benchmark polls the network much more frequently than a real application would do.

The second benchmark shows how the latency depends on contention when using 64 processors (see Figure 4.20). Each processor sends a remote lookup message to a randomly chosen destination, and waits for the reply. We control the frequency of the remote lookup request rate on each processor as indicated on the x-axis. The average latencies vary as shown on the y-axis. On a quiet system, the latency for *Custom* is 22 μs, but it increases to 31.5 μs when the communication traffic is increased to a maximum of over 31,000 lookups per second per processor. In contrast, the latencies for *Native* are twice or three times as high.

Table 4.2 lists the remote lookup performance characteristics for several games on 64 processors, both for *Native* and for *Custom*. The games are discussed in detail in Section 4.7. The second and the fifth column show that use of the customized Myrinet firmware reduces the remote lookup latencies by 44 to 70%. The high latencies of the *Native* versions of checkers and Othello are caused by the high execution times of the evaluation function and the move generator (these are listed in Figure 4.9 and discussed later in this chapter). The network is not polled during execution of these functions, thus lookup request messages are not serviced while the server is expanding or evaluating a node. The latencies could be decreased by a few tens of microseconds by carefully inserting poll statements in the evaluation function. This is hard to do within the move generator, since the move generator is generated by the Multigame front-end compiler. The front-end compiler has no clue about when to insert poll statements in the generated code: inserting too few polls leads to high remote lookup latencies; inserting too many polls leads to unnecessary polling overhead. The third and sixth column give the number of remote lookups per second per processor. The

game	*Native*			*Custom*		
	time (μs)	lookups/s	app. (%)	time (μs)	lookups/s	app. (%)
chess	**64.9**	9,150	59.3	**26.1**	15,071	39.3
checkers	**86.6**	5,150	44.6	**26.0**	7,482	18.6
Othello	**79.2**	5,587	44.3	**24.1**	8,456	20.2
15-puzzle	**53.2**	15,597	83.0	**29.4**	26,501	78.2
double-blank	**55.0**	15,125	83.2	**31.1**	25,478	78.9
Rubik's cube	**58.7**	11,847	69.5	**26.8**	21,114	56.6

Table 4.2: Performance of the partitioned transposition table with and without customized firmware for six games on 64 processors.

fourth and seventh column show the fraction of time spent performing remote lookups at the application level.

Discussion

Our current implementation of *Custom* is based on the experiences we described in [16]. We list a number of design differences with respect to our previous implementation.

Custom is an extension to LFC, rather than Illinois Fast Messages [80]. Since LFC itself takes care of flow control (i.e., slows down senders that send data faster than receivers can handle), we leave this issue to a lower-level layer within LFC, simplifying our *Custom* implementation.

We deal with concurrency control between the CPU and the NP on the server side differently. Instead of using expensive Peterson's locks, we use an optimistic approach.

We simplify the network firmware by not customizing remote update messages for partitioned tables. Little performance could be gained by doing so, since remote update requests require CPU intervention on the server side anyway (the transposition table replacement algorithm is too complex to be handled by the slow LANai NP), and is asynchronous, thus less critical to performance than the synchronous lookup requests.

We do not customize the firmware for *replicated* tables. LFC already contains specialized firmware for doing spanning tree broadcasts on the NPs [17]. LFC implements broadcast efficiently: on 64 processors, each processor is able to broadcast 240 KB/s. Rather than dispatching an incoming broadcast message to the CPU, customized network firmware could copy the new entries in the broadcast message directly into the transposition table (see also Figure 4.17). This implies that the NP has to do many virtual-to-physical address translations (in software rather than through a Memory Management Unit), and set up many small DMA transfers. Since this increases the workload on the NP, it will decrease the broadcast bandwidth. The workload on the CPU decreases, but if communication bandwidth is already the bottleneck, the application will use the extra CPU cycles only to increase the pressure on the network even more. It is not clear whether customizing network firmware for replicated tables can improve remote update performance compared to native LFC.

We decided to extend LFC rather than writing a new LANai control program from scratch. The advantages are twofold: extending an existing control program is much simpler, and the application can use normal communication primitives (e.g., for work distribution) as well. The disadvantage is performance loss, since the control program carries native code that needs to be executed as well. Moreover, the design of the native control program hampers optimizations to the customized extensions. As an example, we could have used application semantics to implement flow control instead of using LFC's flow control scheme, since the application can guarantee an upper limit on the number of outstanding lookup requests. By reserving enough dedicated receive buffers for lookup request messages in NP memory, receive buffer overflow will not occur and flow control will not be necessary. However, as already stated, implementation of such a scheme would require a significant effort.

Related work

The performance of our *Custom* implementation for partitioned tables compares well to other message-passing interfaces built on top of Myrinet. BIP [90, 91] reports the lowest one-way message latency: almost 5 μs for a single-word message. On our system we measured for BIP a minimum roundtrip time of 14.0 μs for a single-word message, and a 17.0 μs roundtrip time for a request-reply pair that matches the message size required for doing a remote lookup. This does not include the time to copy a remote table entry to the reply message.

There are two factors that make BIP fast. First, the LANai control program of BIP is very simple. The LANai control program of *Custom* includes all LFC functionality, such as the ability to multicast messages, and the flow control mechanism for messages other than transposition-table lookup and reply messages, that prevents senders from injecting messages faster than a receiver can handle. BIP leaves flow control to the application level. Second, BIP assumes that the CPU of a receiver is polling the network before a sender starts sending a message, at least for large messages. For irregular applications like game-tree search, this is impossible to guarantee without loss of performance. For a small message like a lookup request, it is not necessary for the receiving (server-side) CPU to react immediately, but the remote lookup latency will increase if the CPU reacts slowly, because the NP–CPU interaction on the server side is still in the critical path.

Other work on customizing network interface firmware has focussed on optimizing message-passing performance for general-purpose message-passing libraries [33, 40, 41, 80, 118], and can result in large performance improvements. Existing systems layer shared data structures on top of a general-purpose message-passing layer. The main problem with most message-passing layers is that they do not allow remote memory to be accessed without transferring control to a remote process. We take a different approach, and use the network interface to avoid control transfers and to perform simple, but application-specific tasks.

Conclusions

From the study above we draw the following conclusions. Customizing the firmware on the network interface processor can significantly improve the performance of remote operations on application data structures, especially if the remote operations are synchronous. By removing the NP–CPU interaction on the server side of a remote partitioned transposition table lookup operation, we reduce the remote lookup latency by 44 to 70%, and improve application performance by 31 to 47%. The drawback of this approach is the high programming effort required to program the LANai processors on the Myrinet interface boards. The lack of suitable debugging tools and the ease with which a misbehaving LANai control program crashes the entire machine make it hard to program the LANai NPs. Moreover, we consider the inability of both the CPU and the NP to perform an indivisible read-modify-write operation on one another's memory as a deficiency. Such a feature would decrease the costs to synchronize the CPU and the NP for access to shared data, rendering dangerous and application specific unsynchronized access methods like described above unnecessary.

4.4.5 Prefetching

The main disadvantage of partitioned tables is that remote lookups are synchronous, and that the processor that issues a remote lookup request idles until the reply message arrives. Even using the customized network firmware, a remote lookup takes between 22 and 32 μs on our hardware, depending on the number of processors and on the contention on the network. To minimize the idle times, we study a second optimization: prefetch transposition-table entries for nodes that are likely to be searched in the (near) future. To hide memory latencies modern microprocessor architectures such as the SPARC V9 [119], the IMPACT EPIC [7], and the IA-64 [61] allow explicit prefetching of memory reads.

For our partitioned implementation, we added a single procedure to the interface of the transposition table, called *TransPrefetch(Node)*. A search engine can call this procedure whenever it is likely that a *TransLookup(Node)* will be performed in the near future. The interface is simple and elegant. *TransPrefetch* can be called anywhere, at any time, and need not necessarily be followed by a *TransLookup*. This is useful if a search engine needs to prune children: a child can be pruned after a prefetch, but before it would have been looked up.

Internally, the prefetch mechanism works as follows. *TransPrefetch* checks if the node's transposition table entry is remote. If this is the case, it sends a lookup request message to the processor that stores the entry. This request is sent *asynchronously*. The procedure then returns. The function *TransLookup* is modified to check first whether a prefetch has been issued for this node. If not, it sends the lookup request message. In either case, it receives the lookup reply message, and blocks if the reply is not yet there.

If the search engine issues a prefetch but prunes the node rather than calling *TransLookup*, the lookup reply message (which is the result of the prefetch) will not be automatically received and cleared. To clear the receive buffers, a ring buffer of outstanding lookup request messages is maintained. A current ring buffer pointer is advanced after each prefetch. If it finds, after an entire cycle around the ring, that the reply corresponding to the request still has not been read by the application, the reply is thrown away. If the search engine still calls *TransLookup* at a later moment, the prefetch simply fails, and a normal, blocking remote table lookup is performed. A beneficial consequence of using a ring buffer is that prefetches which are issued too long before the actual lookup time out; this prevents the search engine from using old data.

Whether the prefetch mechanism is beneficial, depends on several factors. First, if the prefetch is issued less than the roundtrip time before the actual lookup, the latency is not completely hidden. Second, if the prefetch is issued long before the actual lookup, either the prefetch fails due to a ring buffer timeout (the chance that this happens depends on the size of the ring buffer), or the chance increases that stale data are used. Third, prefetching is not free: the ring buffer adds a little overhead; however, this overhead is small compared to the roundtrip latency. Fourth, a search engine that starts prefetching at random (or, as we tried, as soon as a child is created), quickly congests the network, thus a more conservative prefetching scheme is needed.

The two-player search engines use the following rules for prefetching. Before searching child c, child $c+1$ is prefetched. If child c causes a cutoff, the prefetch for child $c+1$ is disregarded. Note that in case of a cutoff, the search engine often continues with the parent's brother, for which a prefetch is also pending. Since a first child has no predecessor, it cannot be prefetched this way. Unfortunately, there is little time between the moment that it is known which of the children is the first (most promising) one, and the moment that the first child is searched. Only if the transposition lookup of the *parent* succeeds *and* contains a "best child" entry, is it known in advance which child will be the first one, and a prefetch is issued as soon as the child is created. Interior nodes usually have this information available from previous search iterations. However, at the leaves of the tree (where most time is spent), the parent will often not have this information available, since the leaf nodes are normally visited for the first time.

The IDA* search engine does not order children, and therefore prefetches the first child as soon as it is created, even before the remaining children are generated. The remaining children are prefetched the same way as the two-player search engines do.

We will now analyze the performance of this prefetching mechanism, and show where prefetching is successful and where it is not. Figure 4.21 shows prefetch char-

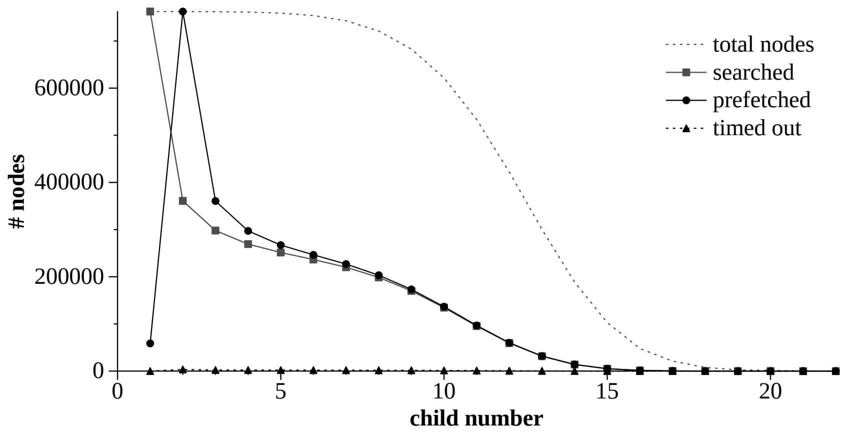


Figure 4.21: Prefetch characteristics for Othello.

acteristics for Othello,[3] using the MTD(f) search engine. To understand the performance of the prefetching mechanism, it is necessary to distinguish several classes of children. After the children of a node are created by the move generator and ordered by the heuristics, the most promising child belongs to class "1", the second most promising child in class "2", and so on. The characteristics in the figure are given for each separate class (e.g., the "1" on the x-axis shows the data for all first, most promising children).

The curve "total nodes" shows the total number of children in each class. Many of the children with child number "2" and higher are pruned by the MTD(f) algorithm; the number of children that is actually searched is given by the curve "searched". From the figure we see, for example, that approximately half of the second children are pruned. The curve "prefetched" shows how many prefetches occur for each class. Only 59,000 out of 762,000 first children are prefetched; often it is not possible to predict which child is the first child far in advance; no prefetch is issued then. For the remaining classes, too many children are prefetched. All second children are prefetched, whether pruned or not. Prefetching is most effective for the higher numbered classes, since pruning is less common there. The curve "timed out" gives the amount of children for which both a prefetch and a lookup are done, but for which the prefetch timed out; this happens close to the root. Prefetched children that are pruned also time out; these are not shown by this curve. With a ring buffer size of 64, the number of timeouts is negligible.

Taking all classes together, we obtain the following success rates for Othello: 77% of the searched nodes are already prefetched, and 82% of the prefetched nodes are

[3]We show numbers for Othello, since the average branching width is not blurred by quiescence search.

game	no prefetch		prefetch			
	time (μs)	app. (%)	time (μs)	overhead (μs)	total (μs)	app. (%)
chess	**26.1**	39.3	16.6	4.53	**21.5**	34.0
checkers	**26.0**	18.6	14.0	4.74	**18.1**	14.0
Othello	**24.1**	20.2	9.33	4.10	**13.3**	12.3
15-puzzle	**29.4**	78.2	18.5	3.68	**22.1**	72.5
double-blank	**31.1**	78.9	21.6	3.65	**25.2**	75.1
Rubik's cube	**26.8**	56.6	8.00	3.64	**11.6**	36.3

Table 4.3: Prefetch characteristics for six games on 64 processors.

indeed searched.

Table 4.3 shows prefetch characteristics for six games, measured on 64 processors. For this experiment, the search engine performs a remote lookup for each node it searches, even at the leaves. The second column gives the average remote lookup times when no prefetching is done. The third column indicates the fraction of the total runtime the application is busy looking up remote transposition table entries. The fourth column lists the average remote lookup times with prefetching, and the fifth column shows the overhead of a prefetch itself (mainly the time to issue a remote lookup request). One cannot add the sum of the latter two to obtain the total cost of doing a remote lookup, since some of the lookups are not prefetched and some of the prefetches are not looked up. The average total cost to do a lookup is given in column six. Like the third column, the last column gives the fraction of the total time spent doing remote lookups, but now with prefetching enabled.

The numbers show that prefetching is always beneficial; the remote lookup times (including the prefetching overhead) decrease for all games. However, for Rubik's cube and Othello, the difference in performance is greater than for the other games. This is explained by the average branching factor; Rubik's cube and Othello build wider trees than the other games.[4] Prefetching works better for wider trees, because a larger fraction of nodes will be successfully prefetched. Remote table lookups are still expensive for the 15-puzzle and the double-blank puzzle. The evaluation functions of these games are so fast that the latency of a remote table lookup cannot be fully hidden.

In the discussion above, we implicitly assume that all children of a node are generated at once. This is true for Multigame, but other (two-player) game-playing programs often generate children one at a time, at the moment that it is to be searched. If such a program should prefetch transposition-table entries, it is necessary to (speculatively) generate child $c+1$ and issue a prefetch before child c is searched. If child c generates a cutoff, a small penalty is paid for needlessly generating child $c+1$;

[4]For chess, the average branching factor is about 35, the widest among the tested games. However, at the bottom of the tree, where most time is spent, quiescence search makes the tree much more narrow.

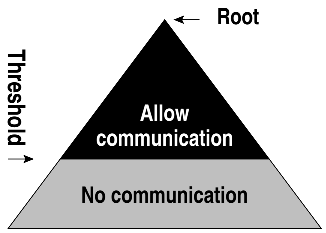


Figure 4.22: Communication threshold.

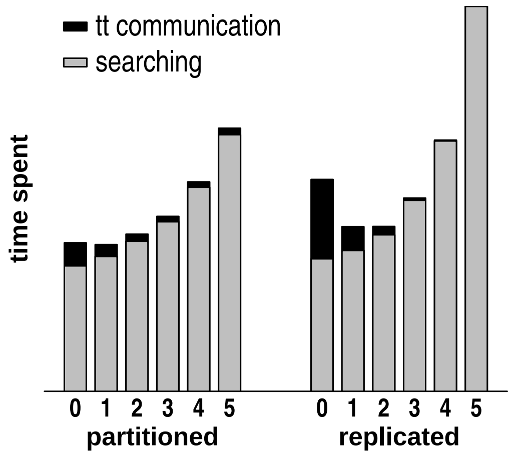


Figure 4.23: Varying the communication threshold for Othello.

otherwise $c+1$ had to be generated anyway, and there is no additional cost.

The literature describes other ways to deal with remote lookup latencies. *Zugzwang* [49] concurrently sends a lookup request for a node and speculatively generates the same node. This partially hides the remote lookup latency. **Socrates* [62] performs synchronous lookups and accepts a 7% execution penalty by waiting for the lookups. On future hardware such an approach may not be acceptable, since improvements in network latencies do not keep pace with the increase of CPU power.

We draw the following conclusions. For all six games we tested, prefetching is beneficial. Prefetching reduces the latency for a remote lookup by 19 to 57% compared to a synchronous lookup, including the overhead to manage the ringbuffer. On application level, the performance improves by 4 to 25%. Prefetching works best when the search tree has a high branching factor near the leaves, as is the case with Rubik's cube and Othello. For applications with narrow subtrees near the leaves, the performance gains are small.

4.4.6 Selective table accesses

The frequency of transposition-table accesses depends much on the search characteristics of a game. If only a small amount of time is spent in the evaluation function and move generator, each processor can search in the order of 100,000 nodes per second on our hardware, accessing the transposition table with a very high frequency. Even with a fast network, there are too many remote accesses per second for both replicated and partitioned transposition tables. This severely increases search times, since processors are slowed down too much by the increasing communication latencies and the

table type	lookup	update	size	optimizations
non-shared	local	local	$p \times m$	
replicated	local	broadcast	m	msg. combining, selective updates
partitioned	RPC	async. send	$p \times m$	custom firmware, prefetching, selective lookups

Table 4.4: Summary of transposition-table distribution characteristics. p is the number of processors; m the amount of memory per processor.

large number of incoming messages that must be handled.

An obvious technique to decrease the communication overhead is not to access the distributed transposition table when searching near the leaves of the tree [101] (see Figure 4.22), and apply table lookups and updates selectively. The possible gains of a hit near the root of the tree are larger, because a pruned subtree rooted high in the tree saves more time than a small subtree rooted low in the tree. For the same reason, move ordering near the leaves is not as important as move ordering near the root. If the costs of a remote lookup are larger than the expected gains, one should search the subtree without doing the remote lookup. For *partitioned* the lookups are relatively expensive, so we vary the depth to which we allow a lookup. For *replicated*, the updates are expensive, so we vary the depth to which we allow an update.

Figure 4.23 shows the average execution times for searching Othello trees on 64 processors for several lookup thresholds (for *partitioned* tables) and update thresholds (for *replicated* tables). The figure illustrates that the communication overhead can be traded for the search overhead, and that there is an optimum that minimizes the total search time. For the performance measurements in the remainder of this chapter, we determine for each game and each number of processors the optimal lookup threshold for *partitioned* and the optimal update threshold for *replicated*. The thresholds are given in Section 4.7.

4.4.7 Summary

In this section, we discussed three transposition-table distribution techniques: *non-shared*, *partitioned*, and *replicated* tables (see also Table 4.4). The implementations of the *partitioned* and *replicated* tables each use their own set of optimizations to reduce the communication overhead.

For *partitioned* tables, we customize the network firmware to reduce the remote lookup latency by a factor 1.8 to 3.3. Prefetching yields an additional reduction by a factor between 1.2 and 2.3. Moreover, the application can skip lookups near the leaves whenever the costs outweigh the potential gain. By optimizing the remote lookup latency, the remote lookup threshold can be kept close to the leaves, reducing

Heuristic	One-player games	Two-player games
Transposition table	$\surd$	$\surd$
History heuristic	–	$\surd$
Quiescence search	–	$\surd$
Repetition detection	$\surd$	$\surd$
Pattern databases	$\surd$	–

Table 4.5: Game-independent heuristics.

search overhead.

For *replicated* tables, we use the weak coherency demands of the transposition table and combine messages, which are broadcast at once. We rely on the broadcast bandwidth of *LFC* to obtain a high update throughput. Unfortunately, the amount of time-consuming message handler invocations scale quadratically with the number of processors, and we expect *replicated* to perform poorly on large-scale systems. Increasing the update threshold to a level further from the leaves reduces the communication overhead, but increases the search overhead.

Application performance results and analyses of the three transposition-table implementations and their optimizations are postponed until we discussed the other runtime system components. The performance of the transposition tables is analyzed in detail in Section 4.7. We will draw conclusions afterward.

4.5 Other heuristics

In this section, we describe the other heuristics we use to improve the search performance of the Multigame runtime system. We describe the following heuristics: the *history heuristic*, *quiescence search*, *repetition detection*, *pattern databases*, and some game-dependent heuristics used in the 15-puzzle.

Table 4.5 lists the game-independent heuristics, and for which kind of games the heuristics can be used. The *transposition table* prevents the search engine from searching the same subtree multiple times, and is used to order the children. Another move-ordering heuristic is the *history heuristic*; this heuristic is used in two-player games only. *Quiescence search* extends the search at positions where the evaluation value is unreliable; this heuristic also is used in two-player games only. *Repetition detection* recognizes cycles in the directed search graph, which is useful in both one-player and two-player games. *Pattern databases* improve the evaluation function in one-player games.

4.5.1 The history heuristic

The history heuristic [102] is a game-independent heuristic for move ordering. The heuristic assumes that the best move from one position is often the best move from a similar position. For example, moving a piece from c7 to c8 may often be the best move, regardless of the surrounding pieces. The history heuristic records how frequently a possible move is the best move, and uses this information for move ordering. The killer heuristic [110] is a special case of the history heuristic [102], therefore the Multigame runtime system does not implement the killer heuristic. We did not implement the heuristic for use by one-player games, since move ordering in one-player games is hardly beneficial [94].

Chess and checkers-playing programs normally implement the heuristic by using two matrices, one for each player. The matrix is indexed by the field number *from* which a piece is moved and the field number *to* which a piece is moved. After the children of a node have been generated, the table entries for each move are read from the table, and the children are sorted in descending order. The most promising move is searched first. After the children have been searched, the matrix entry corresponding to the best move (or the move that caused a cutoff) is incremented. Usually, better results are achieved when the increment value is 2^d where d is the depth to which a node has been searched, instead of using an increment value of 1.

With deep searches, the entries in the table occasionally overflow when 2^d is used as the increment value. We observed that ignoring the overflows can considerably decrease the search performance. Whenever an overflow occurs, all entries in the table are divided by two; subsequent increments are divided correspondingly.

For a general Multigame program it is not possible to implement the heuristic by using a matrix, because legal moves are not restricted to moving exactly one piece from one field to another. Therefore, we use another approach. We map each move to a *signature difference*. The signature difference is the bitwise xor (exclusive or) of the signature of the position *from* which a move is made and the signature of the position *to* which a move is made. The signatures are the same signatures as used for transposition-table hashing. Due to the way signatures are computed [122], a particular move always yields the same signature difference, irrespective of the pieces on the other fields. The history table contains all signature differences that occurred in a tree traversal; usually a few thousand. Associated with each signature difference is the score that indicates how good the move is. Access to the list is sped up by hashing.

An implementation based on signature differences behaves different from one based on matrices. In the former case, a queen move and bishop move from d2 to f4 are considered different moves, while the latter case makes no distinction between them. It is not clear whether this has any impact on performance.

The transposition table is used for move ordering as well. If the transposition table suggests a best move from a certain position, this move is considered first regardless of the contents of the history table, since the information in the transposition table is more reliable. However, the transposition table suggests at most one move; the

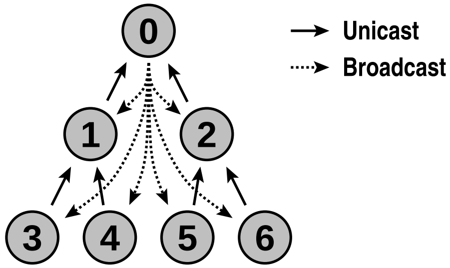


Figure 4.24: History synchronization messages.

remaining children are ordered according to the history heuristic.

For distributed search, it is desirable to share the history table between processors. However, the table is accessed frequently and on distributed memory machines it is too expensive to communicate all accesses. Therefore, all accesses are done locally, and the tables are synchronized only after each search iteration (i.e., each time MTD(f) starts searching with a new search bound, or Alpha-Beta and NegaScout start searching with an increased search depth).

History synchronization is performed as follows. All processors are ordered in a binary tree, as shown in Figure 4.24. When table synchronization begins, each leaf processor sends a message to its parent. This message contains a "delta history table": each entry in the message contains the value with which the corresponding entry in the local history table was incremented since the previous message was sent, thus if the first entry in the previous message contained a 5 while the current first entry in the history table is 7, a 2 is sent in the current message. Other processors receive the data from their children, combine the entries with their local differences, and send the accumulated data to their respective parents. Processor 0 (the root processor) receives all differences, adds them to its local history table and broadcasts the new table to the other processors. By using tree-wise propagation, we prevent one processor from having to receive and process all messages. Moreover, processing is partly done concurrently.

We synchronize the tables only before starting a new search iteration (as suggested by Schaeffer [101]); this happens infrequently. We observed a performance improvement of 21% in search time on 64 processors for chess with respect to a non-synchronized history table. The performance improvements for checkers and Othello are 14% and 10% respectively.

The data distribution for the history heuristic differs from the distribution techniques used for the transposition table. We do not partition the history table for several reasons. First, the history table is much smaller than the transposition table, thus with respect to memory usage, partitioning the history table would not be a real advantage over replication. Second, the access pattern for the history table differs from

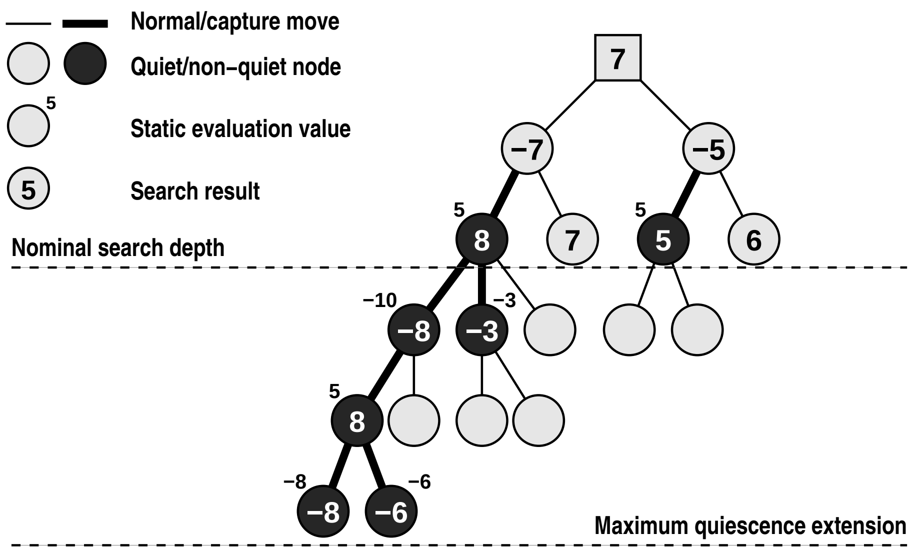


Figure 4.25: Quiescence search.

the access pattern for the transposition table. The read/write ratio for the history table is higher, because *each* child of a node causes a history table lookup (even if it is pruned), and only one of the children (the best one, or the one that causes a cutoff) is involved in a table update. The high read/write ratio does not favor a partitioned history table. Moreover, it is not possible to combine the remote lookup of a transposition table entry with the remote lookup of a history table entry: both heuristics use different hashing schemes and the data are usually stored on different machines.

The data synchronization technique is also different from the replicated transposition table. The replicated transposition table is more often synchronized, and uses all-to-all broadcast. The tree-wise data propagation is more efficient to synchronize the history tables.

4.5.2 Quiescence search

Two-player game trees are searched up to a given depth: the *nominal search depth*. At the leaves of the tree, the evaluation values are taken. If the tree is searched to a fixed depth, an undesirable effect may occur. If the final move between the parent of a leaf node and the leaf node itself is a capture move, the evaluation values of these nodes deviate substantially due to the change in material. In many positions, the opponent can answer a capture with a recapture, but this is not discovered during this phase of the search, because the recapture is beyond the nominal search depth. This is called the *horizon effect* [14].

To circumvent this problem, the search is extended at positions where the evaluation value is not trusted. This search extension is called *quiescence search* [53, 110]. During quiescence search, only tactically disruptive positions are searched, up to a maximum quiescence depth. There are many types of tactically disruptive positions. For example, in chess captures and checks are considered disruptive.

The value of a quiescence node is the maximum of the negated values of the non-quiet children and the static evaluation value of the node. We include the static evaluation value as well (as suggested by Beal [12]), because a (re)capture is not always the best move available.

Figure 4.25 illustrates quiescence search. The nominal search depth is 2, and the search is extended with at most 3 additional plies. The thick lines represent capture moves, leading to non-quiet positions (the gray shaded tree nodes). The leaves on the left hand side (marked with -8 and -6 respectively) are evaluated, because they are at their maximum quiescence depth and are both positions reached by capture moves. Their parent takes the maximum of the negated child results, and the parent's evaluation value. The interior nodes by search results -3 and 5 are non-quiet positions, but none of their children are recaptures.

The two-player search engines in the Multigame runtime system (Alpha-Beta, NegaScout, and MTD(f)) can use quiescence search. We use quiescence search for chess and checkers, but not for Othello. In Othello, *all* moves are capture moves and we do not extend leaf nodes for this game (it can be argued that tactical disruptions at the corners of the field should be extended, as is done in Logistello [31]).

If the change in material balance were the only possible criterion, the decision whether a move is quiet or not would have been game-independent, and could have been implemented by the runtime system itself. However, the Multigame programmer may wish to use additional extension criteria, such as check evasions in chess. Therefore, if quiescence search is desired, the programmer must provide a function that decides if the move between two positions is quiet or not. The interface for this function is:

```
FUNCTION IsQuiet(Parent, Child : POINTER TO NodeType) : BOOLEAN;
```

The function should return false if the move between the parent and child node is quiet, and true otherwise.

4.5.3 Position repetition detection

In many games, it is possible that a position reappears multiple times while playing a game. Usually, the rules of a game describe how to deal with such a situation. For example, the chess rules state that the game is a draw when the same position with the same player to move and the same moving possibilities (i.e., castling and en passant moves) is encountered three times during a game. Another example is the 15-puzzle, where the rules do allow repetition of positions; however, the IDA* algorithm can safely skip repeated positions to avoid superfluous work.

The runtime system uses a backward scan in the list of moves during the game to detect repeated positions. The Multigame system uses some optimizations to detect or refute repetitions quickly, because each searched node is scanned for repetition, and because a complete scan is expensive.

First, the Multigame programmer is allowed (but not obliged) to specify which moves are conversions (i.e., moves that can never be undone, such as captures and promotions in chess) by supplying the keyword ***irreversible***. The Multigame front-end compiler then decides that *each* move, *some* of the moves, or *none* of the moves are conversions. Positions cannot reoccur beyond a conversion. When all moves from all positions are conversions, it is not possible to encounter a repeated position, so no scanning is done. When some of the moves are conversions, the list of moves done is scanned backward up to the most recent conversion. When none of the moves is a conversion, the entire list is scanned.

Another optimization prevents most needless scans, or terminates a scan quickly. Before the list is scanned for duplicate nodes, a hash table, indexed by some of the lower bits of the current position's signature, is consulted. The indexed value stores the number of positions in the list that have the lower bits equal to the index value. For example, if the value at index ***0x1EF*** equals two, there are two positions in the move list that have their signatures ending in ***0x1EF***; they may or may not be the same positions. When this example is used to check a chess position with signature ***0x22222222222221EF*** for three-fold repetition, the entire scan can be omitted, since the hash table indicates that the position occurs at most twice. The optimization works well. For the 15-puzzle, where about one out of three positions is a repetition, only 0.38% of the positions that are not repetitions result in needless scans.

4.5.4 Pattern databases

One-player search algorithms like IDA* use an admissible evaluation function, i.e., an evaluation function that never overestimates the distance from the position to a target. The better the evaluation function approximates the actual distance, the more efficient the search. A slight increase in average evaluation value can reduce the size of the search tree by an order of magnitude. It is therefore important to estimate the distance from a position to a target as accurately as possible (yet the distance may not be overestimated, otherwise a solution is not guaranteed to be the shortest solution).

A particularly useful estimator in one-player search is the *pattern database*. The pattern database can be used instead of, or in addition to, an evaluation function. Like the evaluation function, it returns an admissible distance from a position to a target. The position maps to an index in the database, and the database value is used as a lower bound. Pattern databases have been used for the 15-puzzle [36, 37] and Rubik's cube [68].

The Multigame implementation of Rubik's cube uses pattern databases for evalu-

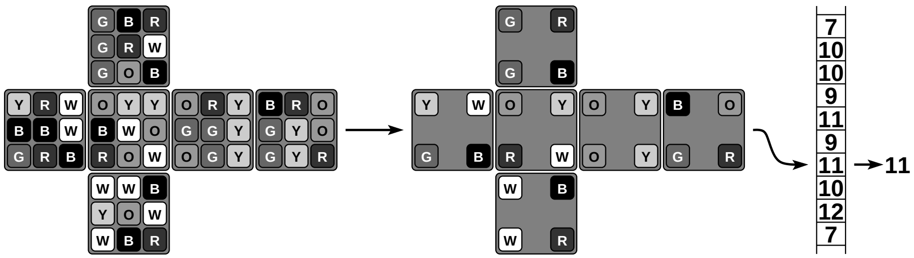
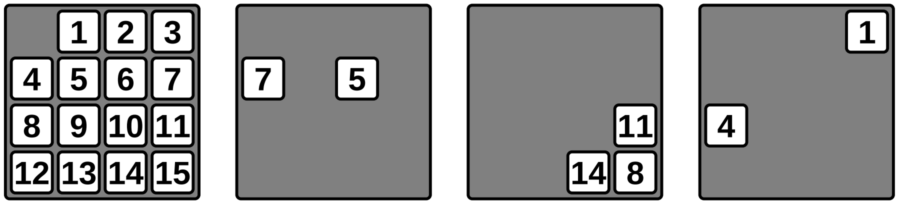


Figure 4.26: Pattern database mapping.

ation of a position, in addition to a simple, traditional evaluation function. The pattern database is used as follows, and is illustrated with an unfolded example position as shown in Figure 4.26. Given the position, some of the cubies are disregarded. For this database, all edge cubies are neglected. The center cubies have fixed colors and are disregarded as well. The remaining corner cubies form a pattern. Both the location and the orientation of the corner cubies are considered in the pattern. The pattern is a simplification of the original puzzle, and can be solved in at most the number of moves required for the original puzzle, and can thus be used as an admissible evaluation value. The database stores solution distances for all pattern permutations. The example pattern is looked up from the database by computing the pattern's index; in this case the database value is 11.

Our implementation of Rubik's cube uses two databases, one 42 MB database for the corner cubes, and one 20 MB database for six of the edge cubes. The twelve edge cubies of a position are subdivided into two patterns of six cubies each, and both patterns are looked up in the edge database. Thus, each cubie is involved once (eight corner cubies plus two times six edge cubies). The maximum of the three values obtained by the database lookups and the value obtained by the evaluation function is taken; this yields an admissible value. However, if the parity of the maximum differs from the parity of the corner database value (i.e., the maximum is even and the corner database value is odd, or vice versa), we add 1 to the maximum (then the maximum and the corner database value are either both even or both odd). This is correct, because the parity of the corner database value always equals the parity of the solution length of the original position.

The Multigame runtime system supports the use of pattern databases through a general interface, thus pattern databases can also be used for other one-person games. Unfortunately, the function that maps an unreduced position via a reduced position to an entry in a database, is game-dependent and database-dependent. The programmer must provide a mapping function in C for each database that is used. The runtime system cannot create pattern databases; a separate program is needed to compute them. Usually this is done using retrograde analysis [8, 52].



(a) target position. (b) linear conflict. (c) corner conflict. (d) last-move conflict.

Figure 4.27: Illustrations of various 15-puzzle conflicts.

In a distributed environment, the database is replicated. The advantage of a replicated database is that each access (read) can be done locally. The disadvantage of replication is the waste of memory. A partitioned scheme allows larger databases and thus reduction in search effort, at the expense of many remote accesses. There are no fundamental reasons to prevent extension of the runtime system with support for partitioned databases. Customizing the network firmware (see Section 4.4.4) to service remote reads directly by the network processor and prefetching would probably reduce the remote read latency by a significant amount.

Pattern databases can be used to trade memory for time. Korf conjectures that the amount of time needed to search a position is approximately inversely proportional to the size of the pattern database searched [68].[5] This would mean that doubling the amount of memory would half the execution time. Holte and Hernádvölgyi empirically verified this conjecture [60] and conjectured a more accurate approximation, which suggests a speedup somewhat less than a factor of 2 when the amount of memory is doubled.

4.5.5 15-puzzle heuristics

The 15-puzzle uses a state-of-the-art evaluation function. It combines the *Manhattan distance*, *linear conflict heuristic* [58], *last-move heuristic* [69], and *corner conflict heuristic* [69].

4.5.5.1 The linear conflict heuristic

The linear-conflict heuristic [58] recognizes tiles that are in the correct row or column, but in the wrong order. An example is given by Figure 4.27(b) (Figure 4.27(a) shows the target position). The Manhattan distance for this position is 4, but it is impossible

[5] Korf uses the A* search algorithm; we use IDA*, which searches more nodes, but has less runtime overhead than A* [28].

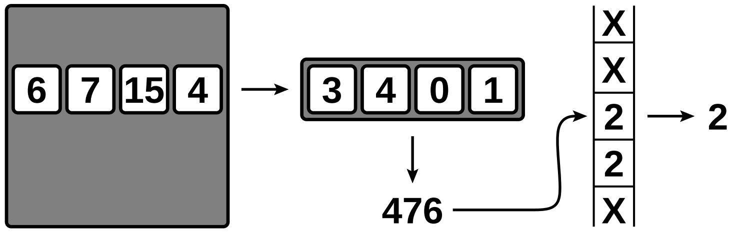


Figure 4.28: Mapping linear conflicts in the 15-puzzle.

to get both tiles in the correct places in 4 moves, since one of the tiles has to move around the other. These pieces form a linear conflict. In this case, the linear-conflict heuristic adds 2 to the evaluation value. The evaluation value is increased by 4 when three tiles in a row (or column) belong to that row, but are mutually wrongly ordered; or even by 6 if all tiles in the row are inversely ordered. When applying the linear-conflict heuristic in the evaluation function of the 15-puzzle, the sizes of the search trees are roughly reduced by a factor of 10, at the expense of a more complex evaluation function.

Linear conflicts are recognized efficiently as follows. First, the tiles in a row (or column) are mapped to pseudo-tiles, as illustrated by Figure 4.28. The example shows the row where the tiles "4", "5", "6", and "7" belong. These tiles are mapped to pseudo-tiles "1", "2", "3", and "4", respectively. All tiles that do not belong to the row and the blank position are mapped to the pseudo-tile "0". Then, the row with pseudo-tiles is read as an integer value with base 5, in this case 3401_5, which is 476_{10} decimal. This number indexes a precomputed array containing the number of extra steps due to a linear conflict.[6] For efficiency reasons, the translation steps in Figure 4.28 use multiple levels of pre-computed arrays. On our hardware, the entire evaluation of a position takes about 2.5 μs.

4.5.5.2 The corner-conflict heuristic

The corner-conflict heuristic [69] looks at the tiles next to the upper right-hand side corner, lower right-hand side corner, and lower left-hand side corner. If the neighboring tiles are at the right places, but the corner tile itself is not (as illustrated by Figure 4.27(c), where tiles "11" and "14" are placed correctly, but "8" is not), one of the neighboring tiles has to be moved to release the wrongly placed corner tile. This increases the evaluation value by 2 per matched corner. The corner-conflict heuristic is not applied to the upper left-hand side corner, because the target position has its blank tile there; therefore no corner conflict can occur in this corner.

[6] An X in the table indicates an illegal pattern, due to multiple occurrences of a single tile in a row (or column).

white *mark –> b2;*	*tic-tac-toe*	draw cross in the middle
a3 –> c5; ***empty field*** *–> b4;*	*checkers*	capture move
e1 –> c1; a1 –> d1;	*chess*	queen-side castle white
empty field *–> e7; queen –> e8;*	*chess*	pawn promotion

Table 4.6: User interface example moves.

4.5.5.3 The last-move heuristic

The sequence of moves that solves a 15-puzzle problem either ends moving the tile labelled “1” from the upper left-hand side corner to the right, or the tile labelled “4” from the upper left-hand side corner down. The last-move heuristic [69] compensates for the fact that the Manhattan distance wants to place the tiles “1” and “4” too fast at their correct places. Figure 4.27(d) illustrates a position with a Manhattan distance of 3. Since one of the tiles has to go through the upper left-hand side corner, the minimum distance can be increased by 2.

The last-move heuristic increases the evaluation value by 2 when tile “1” is not in the left column, and tile “4” is not in the top row. However, when tile “1” or “4” is involved in a linear conflict, the evaluation value is not increased by 2, since this potentially overestimates the distance to the target. For example, when tile “1” is involved in a linear conflict, it must be in the second column (because it has to be in the correct column). Moving it left to the first column both solves the linear conflict and the last-move conflict, thus this move must not be counted twice.

4.6 The user interface

A Multigame program communicates with the player via a shell-like environment. Rather than using a game-dependent graphical interface, the player sees a prompt and can type commands like “read position”, “set search depth”, “search tree”, and “print statistics”. It is, however, possible to write a game-dependent graphical interface on top of the shell. The “play” utility program interfaces between two Multigame programs so that they can automatically play against each other. The shell is almost game-independent; the game-dependent properties it knows about are the dimensions and the layout of the board and the names of the pieces.

The shell is specialized for its game-playing environment. A list of accepted commands is shown in Appendix C, and briefly explains the semantics of each command.

Within a game, moves are described using a simple language. Rather than giving formal detail, Table 4.6 gives a few intuitive examples. The move descriptions are used both as input (for example, via the ***move*** command) and as output (for example, as the result of a ***hint*** command). Sometimes multiple descriptions describe the same move: the shell accepts any of them, or will print one of them. The shell uses the move generator to see whether a given move is legal from the current position by checking whether the position after doing the move is among the children.

The ***statistics*** command is used to print statistics. Each machine maintains statistics (provided that statistics are configured — see Section 4.7.2) and these are collected when the user enters the ***statistics*** command. By default, all statistics collected by all processors are printed, but the user can specify any subset of this, for example, the transposition table and history heuristics statistics on processors 3, 4, and 7–10.

Time control has not yet been implemented in the Multigame runtime system. For tournament play, time control is indispensable. Currently, the search is limited by controlling the (nominal and quiescence) search depth.

4.7 Performance results

In this section we analyze the performance of the Multigame runtime system and study the behavior of three one-person and three two-person games. We also compare the behavior of different distributed transposition table implementations, since the sharing strategy of the transposition table has great impact on the parallel performance of a Multigame application. For a description of the hardware we used for the performance measurements, we refer to Section 1.3.

This section is structured as follows. First, we introduce the applications. Then, we describe the configuration parameters used for each of the applications. Next, we justify the timing methodology. Further, we show the application speedups; the performance of each of the games is subsequently analyzed in detail. Finally, we discuss the results and conclude.

4.7.1 Applications

We study the parallel behavior of six different Multigame applications. The two-person games are *chess*, *checkers*, and *Othello*; the one-person games are the *15-puzzle*, the *double-blank puzzle* (explained in Section 4.7.5.5), and *Rubik's cube*. The two-person games use the MTD(f) search engine and the one-person games use the IDA* search engine. We chose for these applications for several reasons. First, the different search characteristics of the games (e.g., wide and shallow vs. narrow and deep search trees; few vs. many transpositions) provide a representative suite that measures the Multigame software under varying circumstances. Second, we have good evaluation functions available for each of the applications.

The performance of the games is measured using three distributed transposition-table implementations: *partitioned*, *replicated*, and *non-shared*. In this section, we will pay attention to the performance behavior of these transposition-table implementations.

description	compile-time option	chess	checkers	Othello	15-puzzle	double blank	Rubik's cube
multi-threading	√	–	–	–	–	–	–
multi-processor	√	√	√	√	√	√	√
use interrupts	√	–	–	–	–	–	–
search algorithm	√		MTD(f)			IDA*	
nominal search depth	–	7–13	19–27	11–15	–	–	–
iterative deepening step size	–	2	2	2	–	–	–
quiescence search	√	√	√	–	–	–	–
maximum extension depth	√	12	12	–	–	–	–
minimum job depth	√	11	18	5	40	40	9
signature size (bits)	√	64	64	64	64	64	64
Multigame move generator	√	–	√	√	–	√	–
incremental signatures	√	√	√	√	√	√	–
order children	√	√	√	√	–	–	–
Young Brothers Wait	√	1	1	1	–	–	–
transposition table	√	√	√	√	√	√	√
number of entries	√	2^{22}	2^{22}	2^{22}	2^{22}	2^{22}	2^{21}
lookup threshold *partitioned*	√	0–5	0–13	0–1	†	0	0
update threshold *replicated*	√	0–9	0–13	0–1	†	†	†
buffer size *replicated*	√	250	250	250	250	250	250
aging	√	√	√	√	–	–	–
store entire board	√	–	–	–	–	–	–
history heuristic	√	√	√	√	–	–	–
synchronize tables	√	√	√	√	–	–	–
game dependent ordering heur.	√	√	–	–	–	–	–
position repetition detection	√	√	√	√	√	√	√
opening book	√	–	–	–	–	–	–
pattern database	√	–	–	–	–	–	√
evaluation cache	√	–	–	–	–	–	–
maintain statistics	√	√	√	√	√	√	√

† See text.

Table 4.7: Configuration parameters.

CPUs	lookup threshold *partitioned*							update threshold *replicated*						
	1	**2**	**4**	**8**	**16**	**32**	**64**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Chess	0	4	4	5	5	5	5	0	0	0	5	6	7	9
Checkers	0	0	0	12	12	13	13	0	0	0	0	12	12	13
Othello	0	0	1	1	1	1	1	0	0	0	0	1	1	1

Table 4.8: Thresholds for two-person games on various numbers of processors.

4.7.2 Configuration parameters

As already stated in Section 4.1, the Multigame runtime system is recompiled for every game. The compile script accepts a few dozen switches to select which components should be included in the runtime system, and to set various parameters. Most parameters are set at compile time rather than at run time (via the Multigame shell described Section 4.6), to generate more efficient code, at the expense of less flexibility for the player.

Table 4.7 lists a number of compile and run-time arguments, as we used them in our performance measurements. These do not include the arguments that specify the target architecture or control the optimization and debugging levels for the generated executable.

For all performance measurements we disabled *multi-threading* and enabled *multi-processor* support. *Interrupt* driven message delivery was disabled; we use polling instead. The two-player games use MTD(f) as *search algorithm*; the one-player games use IDA*. Each position was searched to a *nominal search depth* between 7 and 13 for chess, 19–27 for checkers, and 11–15 for Othello. For these games, *iterative deepening* repeatedly increases the search depth in steps of 2. For chess and checkers, *quiescence search* extends the search depth with at most 12 plies. These depths correspond to search times of a few minutes on 64 processors. The *maximum job depth* specifies the minimum distance to the leaves (including quiescence extensions) to which remote work stealing is allowed. All games use 64-bit *signatures*. Checkers, Othello, and the double-blank puzzle use *move generators* generated by the Multigame compiler; the others are written in C. The move generator maintains the *signature* incrementally for all games except Rubik's cube; for the latter it is faster to recompute the signature after each move. The two-player algorithms order the children according to the "best move" information in the transposition table and the information in the history heuristic; the one-player algorithms do not order the children. Chess uses game-dependent move ordering as well: captures and checks have a higher priority than other moves. Young Brothers Wait restricts the parallelism to all children except the first one for the two-player games.

All programs use a *transposition table*. To leave room for the pattern databases, Rubik's cube uses a smaller table than the other games use. The *lookup threshold*s for *partitioned* tables are chosen in such a way that the total overhead (search overhead plus communication overhead) is minimized. The optimal threshold depends on the number of processors; extensive experimentation was required to determine them. The thresholds are shown in Table 4.8. The thresholds are the distances to the leaves, including quiescence search extensions. Chess thus performs remote transposition-table accesses even during quiescence search. Checkers does so only on small numbers of processors, but checkers spends only a few percent of the time in quiescence search. How the 15-puzzle restricts the remote lookups is explained later in this section. The double-blank puzzle and Rubik's cube perform lookups for all nodes. The *update threshold*s for *replicated* tables are also chosen to optimize performance. The thresholds are also shown in Table 4.8. Remote update thresholds for the 15-puzzle, double-blank puzzle, and Rubik's cube are explained later. All games use a broadcast buffer for *replicated* that contains up to 250 entries. *Aging* is enabled for two-player games; however, table entries never become aged because we stop searching after one move. None of the games stores the *entire board* into a table entry; instead they store the tag part of the signature.

The *history heuristic* is used by the two-player games to order the children; the history tables are occasionally *synchronized*. All games detect *repetition* of positions. For none of the games, an *opening book* is used. Rubik's cube is the only game for which we use a *pattern database*. A separate *evaluation cache* (to avoid multiple evaluations of the same position) is never used; the transposition table is used for this purpose (in this case, the "best move" field in the table entry is void). Finally, *statistics* are maintained during the search.

Many of the items listed in Table 4.7 were found empirically, by tuning until best performance was obtained. This way of tuning is tedious and time-consuming, especially because the tuning was done on each number of processors separately. A tool that automatically tunes a game on a given system is desirable.

Several approaches to automate the tuning of the system are feasible. A genetic algorithm, where the settings are encoded in a gene, will eventually find best or at least good settings. Hill climbing is another brute-force approach, but will probably require much time to find good settings as well. Most desirable is a dynamic monitoring system within the runtime system itself that monitors the performance at runtime and automatically adjusts settings when it believes that the current settings are suboptimal. For example, if it monitors a high search overhead and a low communication overhead, it could lower the threshold for remote transposition-table accesses. This would eliminate programmer intervention entirely. Dynamic monitoring could also lead to better results when the shape of a tree changes during the different phases in a game, such as chess trees, which are much wider during the middle game than in the end game.

4.7.3 Timing methodology

Each of the applications is measured using an application-specific set of test positions. Each of the positions is tested up to four times for a particular number of processors, and for each position the execution times are arithmetically averaged. Since the run times for different test positions vary substantially, the geometric mean of each position is taken to obtain the total average execution time. The same way of averaging is used for timing the individual program parts, such as the time spent in the evaluation function.

We use the Pentium Pro's time stamp counter to do accurate measurements. This 64-bit counter is incremented each clock cycle. Reading the time stamp counter requires no special privileges, and can be done by a single assembly instruction. Unfortunately, the Pentium Pro requires over 30 cycles to execute this instruction, where it normally executes two instructions in a single cycle, thus reading the time stamp counter is expensive. Yet the accuracy of this timer makes it suitable for timing both long running code sequences and microsecond events.

The Multigame runtime system provides a timer module that exports primitives for starting and stopping a timer, and printing timing statistics. Starting a timer requires three assembly instructions and stopping a timer four.[7] A start/stop pair takes approximately 320 ns on our hardware. The use of timers has little impact on the behavior of our applications; at the application level, the timing overhead is usually about 2% and at most 3.8%.

For *replicated* transposition tables, it is not possible to measure the communication time in behalf of transposition table accesses exactly, because incoming broadcast messages cannot be timed accurately. If we put timers around each message handler, we underestimate the amount of time spent communicating, because this way of measuring does not account for the overhead in lower message passing layers (Panda and LFC). If we put timers around each poll, we overestimate the amount of communication time, because we include the time to handle non-transposition-table related messages as well. Another complicating factor that hampers exact timings of transposition-table communication is that Panda implicitly polls the network when a message is sent. Therefore, it is possible that a transposition-table broadcast message is implicitly received and handled while a message for an unrelated program part (e.g., work distribution) is being sent. We therefore only measure the time spent in the user-level broadcast handler (which we can time exactly), and measured the Panda/LFC overhead separately. Together, this yields performance numbers that are acceptably accurate.

While running the checkers test set, we observed that multiple, parallel runs of the same test position did not always result in the same computed value for the root of the tree. Occasionally the result slightly differed from the answer computed by a single-processor run. This behavior disappears when we let transposition table lookups suc-

[7] Provided that the *%eax* and *%edx* registers are free.

ceed only when the search depth in the table *exactly* matches the required search depth, rather than accepting table entries that have been searched to *at least* the required search depth. We did not observe this behavior for other games; the deep checkers trees with transpositions at varying depths cause this behavior. Although the answers are different, neither answer is wrong. Accepting a table entry that has been searched deeper than required might just lead to a "better" answer. This unfortunate situation leads to different trees being searched by different runs. We considered accepting exact depth matching table entries only for the performance measurements, but this makes the application unrealistically slow. We therefore chose to accept table entries that have been searched to a sufficient depth, and average the run times over a large number of runs to obtain useful speedups.

4.7.4 Application speedups

Figure 4.29 shows the speedups for each of the applications. All speedups are relative to the version that uses a *non-shared* transposition table, since this version is the fastest on a single processor for all games. Since the 15-puzzle has few transpositions, we also show the speedups for a 15-puzzle variant that does not use a transposition table at all. Measurements for the two-person games are performed for up to 64 processors; for one-person games we include numbers for 128 processors for comparison purposes in Chapter 5.

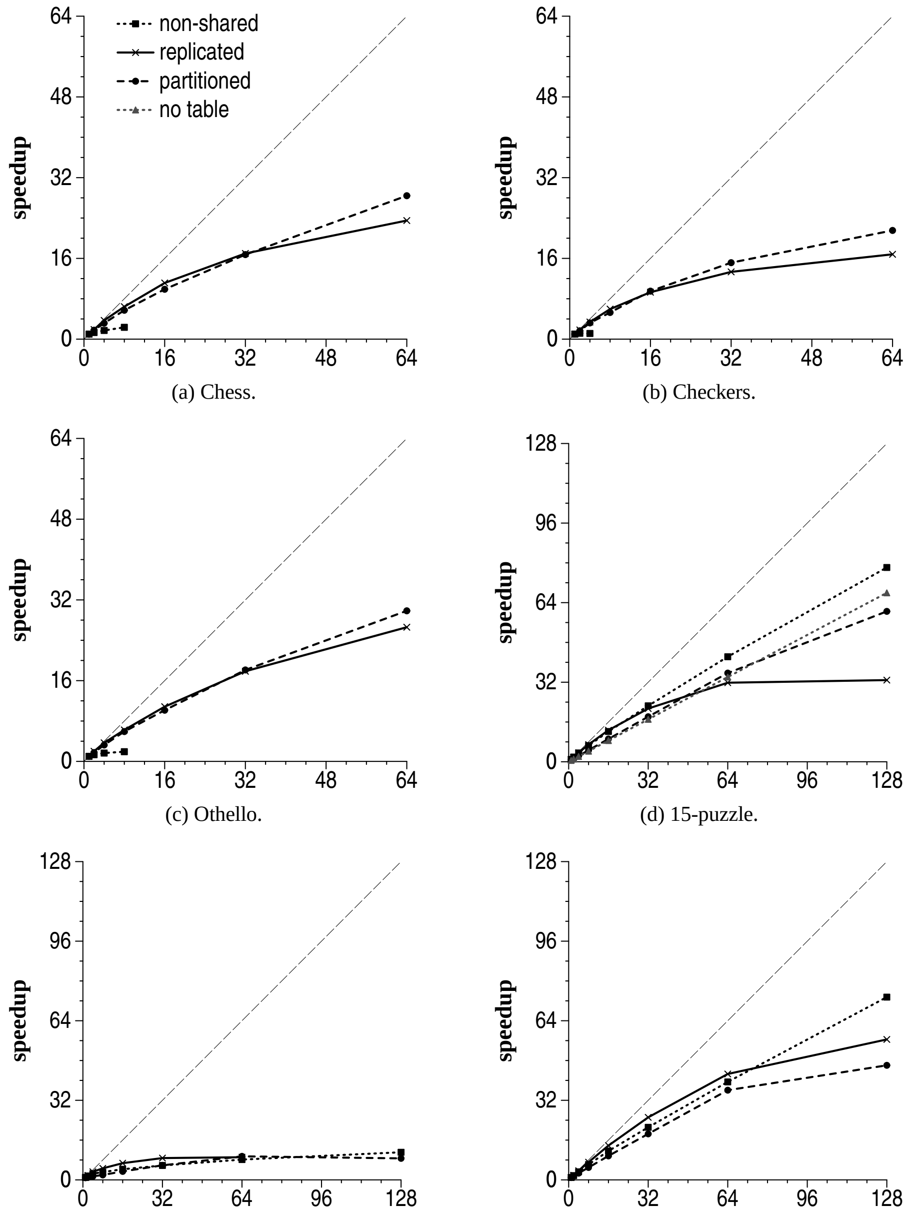
For chess, checkers, and Othello, we did not measure the performance of the variants that use a *non-shared* transposition table on large numbers of CPUs, since these variants do not scale at all. Doing these measurements would require an excessive amount of CPU time.

All figures show speedups for *replicated* that taper off more than the speedups for *partitioned*. For small numbers of processors, *replicated* outperforms *partitioned*, but all figures except Rubik's cube show a crossover point where *partitioned* starts to perform better than *replicated*. In Section 4.4 we explained why *replicated* does not scale to large numbers of processors. *Non-shared* performs well for the 15-puzzle and Rubik's cube, but scales poorly for the other applications.

At first sight, the application speedups seem mediocre. It is well-known that game-tree search is hard to parallelize [4, 49, 101], due to a combination of communication overhead, search overhead, and synchronization overhead. Below we explain the performance characteristics and discuss the various forms of overheads for each of the applications.

4.7.5 Performance breakdown

We now analyze the performance results in detail, to explain the speedups shown in Figure 4.29. The applications suffer from various forms of overheads. To determine the impact of the various overheads, we measured where each application spends its



(a) Chess. (b) Checkers.

(c) Othello. (d) 15-puzzle.

(e) Double-blank puzzle. (f) Rubik's cube.

Figure 4.29: Application speedups for different transposition table strategies.

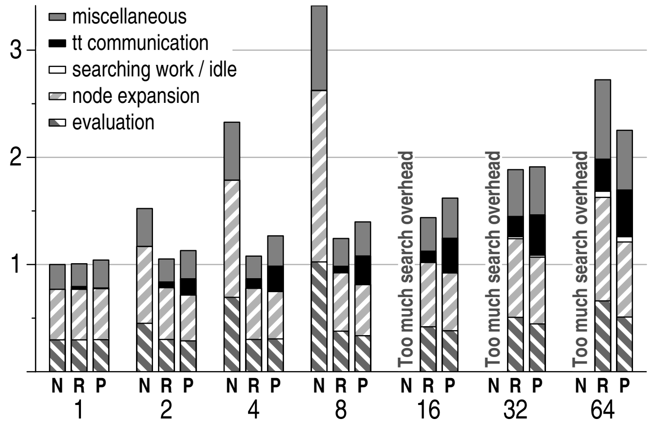


Figure 4.30: Chess performance breakdown.

time and compare these times for various numbers of processors. We report on our findings for each application separately.

We use bar graphs to specify how much time each processor spends in a particular program part (see Figure 4.30 for an example). We distinguish the following program parts:

- *Node evaluation* denotes the amount of time spent in the evaluation function.
- *Database lookups* indicates how much time is spent indexing and looking up databases.
- *Node expansion* specifies how much time is needed for the move generator.
- *Searching work / idle* shows how long processors are busy searching and stealing work from other processors. This includes the idle time due to insufficient parallelism.
- *Transposition table communication* is the time needed for doing remote transposition table lookups and updates, and includes both the time to issue requests and to handle incoming messages.
- *Miscellaneous* is the time spent in the remaining program parts. These include the search engine, move ordering, history heuristic, position repetition detection, node allocation and deallocation, and local job queue overhead.

On the x-axis, the number of processors and the transposition table variant are shown. Here an "N" indicates *non-shared*, "R" means *replicated*, "P" stands for *partitioned*, and "–" (in Figure 4.33) means no transposition table at all.

The height of each bar represents how much time is passed in a particular program part. The total height reflects the total run time. On multiple processors, the height is the sum of execution times obtained on the individual processors. The run times are normalized; the norm is the run time of the *non-shared* variant on a single processor, which is the fastest sequential variant for all games. Everything above the norm is overhead and inversely proportional to the speedup; how the different forms of overhead mutually relate can be determined from the height of the shaded areas. For example, a “2” on the y-axis while using 64 processors corresponds to a 32-fold speedup. The y-axis thus shows normalized accumulative execution times. In some cases, this execution time is extremely high due to search overhead; we omit the results for these cases.

4.7.5.1 Chess

We first look at the behavior of our chess implementation. Both a move generator written in the Multigame language and a move generator written in C exist. Since the C implementation is faster (see Section 3.4), we use the C implementation for our measurements. The evaluation function is ported from CilkChess, a program developed at the Laboratory of Computer Science, MIT. We used the Bratko-Kopec test set [22] (shown in Appendix B.4) to test our chess implementation. The nominal search depth varies from seven to thirteen plies; quiescence search extends this depth up to twelve plies. Our chess implementation does not check the 50-move rule (see Section 3.7).

Figure 4.30 shows how the application spends the time in different program parts for *non-shared*, *replicated*, and *partitioned* transposition tables. The figure shows why the speedups are far from perfect. The application either suffers from a large search overhead (indicated by increasing times spent expanding and evaluating nodes), or both from search overhead and communication overhead (indicated by the increasing times for transposition table communication).

On multiple processors, *non-shared* suffers from a high search overhead. Up to 32 processors, *replicated* performs better than *partitioned*, which is a surprising result. The communication overhead for *replicated* is small for up to 16 processors, while the communication overhead for *partitioned* is apparent even for small numbers of processors.

On 64 processors, *partitioned* performs best. Here, *partitioned* and *replicated* have approximately equal communication overheads, but the search overhead for *partitioned* is smaller, partially due to a lower lookup threshold than the update threshold for *replicated*, and partially due to the larger number of table entries. However, both the search and communication overhead are significant, hampering good speedups.

Our speedups compare well to other work described in the literature. Brockington reports a speedup of 18 on 64 processors for the Crafty chess program, using the APHID parallel search algorithm [27]. Weill achieves a speedup for ABDADA search of 16 on 32 processors [120]. However, comparing speedups is probably useless for numerous reasons. First, we use the MTD(f) search algorithm, while others

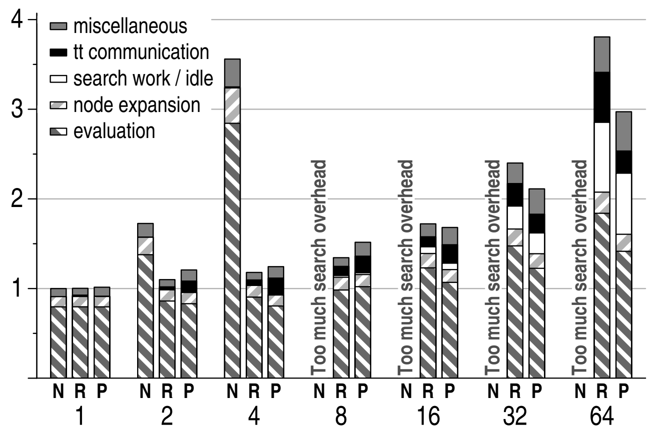


Figure 4.31: Checkers performance breakdown.

usually parallelize NegaScout. As far as we know, so far no published results for parallel MTD(f) exist. The newest version of CilkChess is based on parallel MTD(f), but performance results were not reported. Second, results obtained by others do not easily compare due to differences in hardware. Our interconnect is relatively faster than on most other distributed memory machines, but one to two orders of magnitude slower than shared memory. This allows us to share the transposition table to reasonable depths, at the expense of considerable message passing overhead. Third, the behavior of the sequential implementation has impact on speedups. For example, the efficiency of the move ordering heuristics and parallel search overhead are tightly related (perfect ordering implies zero search overhead). Also, our sequential implementation is somewhat slower than a native chess program (and therefore easier to parallelize), because our move generator fully generates all children of a node while a native chess program could generate a list of legal move descriptions and generate only those positions that need to be searched on demand.

4.7.5.2 Checkers

The second game in our test suite is checkers. The move generator is generated by the Multigame front-end compiler. The evaluation function is ported from Chinook [104, 106], the current man-machine world champion. However, our checkers implementation does not use the endgame databases used by Chinook. The checkers test positions are shown in Appendix B.5, and is the same test set as used by Plaat [86]. The checkers trees are searched between 19 and 27 plies, and quiescence search adds at most 12 plies.

Figure 4.31 shows where checkers spends its time. The figure shows some differ-

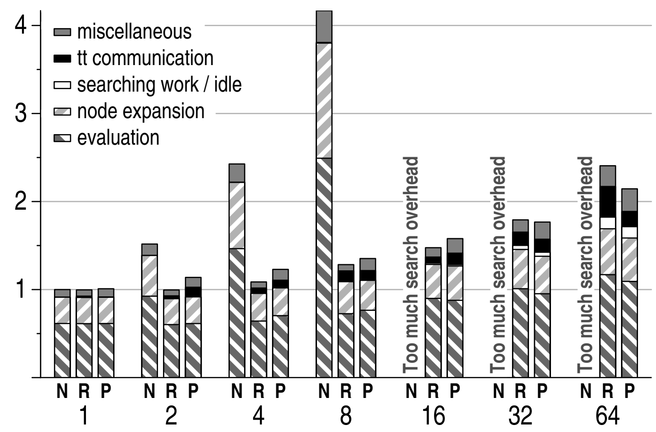


Figure 4.32: Othello performance breakdown.

ences with respect to chess. First, most time is spent in the evaluation function, where chess spends most time in the move generator. Second, sharing the transposition table is even more important than in chess; *non-shared* does not obtain any speedups at all. Third, with checkers, it is difficult to keep many processors busy, resulting in high idle times. The narrow search trees are highly sequential and the small amount of parallelism forces us to allow the transfer of small jobs, increasing the work-stealing overhead. Kuszmaul [71] studied the amount of parallelism in search trees in detail, although he used chess trees for his study, which are much wider and easier to search in parallel than checkers trees. The Multigame runtime system does not instrument the length of the critical path (the execution time when an infinite number of processors is used), nor does it measure the average amount of parallelism.

Replicated performs better than *partitioned* on up to 8 processors; on 16 processors and more, *partitioned* is faster. *Partitioned* profits from the increased number of transposition table entries. When we run the same test set on 64 processors using a version of *partitioned* that has as few table entries as *replicated*, both versions search nearly the same number of nodes. The communication overhead for *replicated*, however, is higher.

4.7.5.3 Othello

The third application we study is Othello. The move generator is written in the Multigame language, and the evaluation function is ported from Aïda, a program developed at the University of Leiden, the Netherlands. The Othello test suite is from Plaat [86], and is shown in Appendix B.6. The positions are searched to 11 or 13 plies. Since there are no quiet positions in Othello, we search the trees to a fixed depth.

The performance breakdown for Othello is shown in Figure 4.32. Othello yields better speedups than chess and checkers, because the search and communication overheads are lower. The search trees are more balanced than chess and checkers trees. Moreover, the work stealing overhead is modest. Although Othello trees have few transpositions, the transposition table is important for move ordering and for maintaining state between iterations of the MTD(f) algorithm. If the table is not shared, the speedups are poor. Up to 16 processors, *replicated* is faster than *partitioned*; starting from 32 processors *partitioned* outperforms *replicated*. On 64 processors, *partitioned* obtains a speedup of 28.4 with respect to *non-shared* on single processor.

4.7.5.4 The 15-puzzle

Our implementation of the 15-puzzle uses a state-of-the-art evaluation function, which is discussed in Section 4.5.5. For the performance measurements we use the move generator that is written in C (rather than the one written in the Multigame language).

The test positions used for the 15-puzzle are shown in Appendix B.2 and are nine of the hardest positions known [52]. All positions have a solution length of 80. Most parallel 15-puzzle programs are benchmarked on the 100 test problems in [67]. Using a sophisticated lower bound and a fast processor means that many of these test problems are solved sequentially in a few seconds. Hence, a more challenging test suite is needed.

The 15-puzzle uses the IDA* search algorithm. To avoid long sequential searches, we stopped searching after a 74-ply search iteration. By stopping the search before a solution is found, we circumvent another problem. The last iteration (in which a solution is found) needs an unpredictable amount of search time, since a solution can appear anywhere in the tree. All previous iterations build the same trees, which require the same search effort. By not searching the last iteration, we obtain reproducible execution times.

Both *replicated* and *partitioned* reduce the amount of transposition-table communication by avoiding remote accesses at the leaves, but the threshold is variable. *Replicated* performs an update for a node when it searched at least 64 nodes in the subtree below it. For *partitioned* such an approach to reduce lookups is not possible, because the lookup occurs before the subtree below it has been searched, and at the time of the lookup the size of the subtree is not known. We therefore use the following heuristic: a lookup for a node is done if the lookup for the parent or the lookup for the grandparent was successful. If neither lookup was successful, the node probably has not been visited by a previous iteration of IDA*, and it is likely that the node is somewhere near the leaves. Using this heuristic increases the number of visited nodes by 31%, but saves a factor 5.3 on communication costs.

Figure 4.33 shows the performance breakdown for the 15-puzzle. Since the 15-puzzle has few transpositions and does not need the transposition table for move ordering, we include the numbers for a version without any transposition table at all.

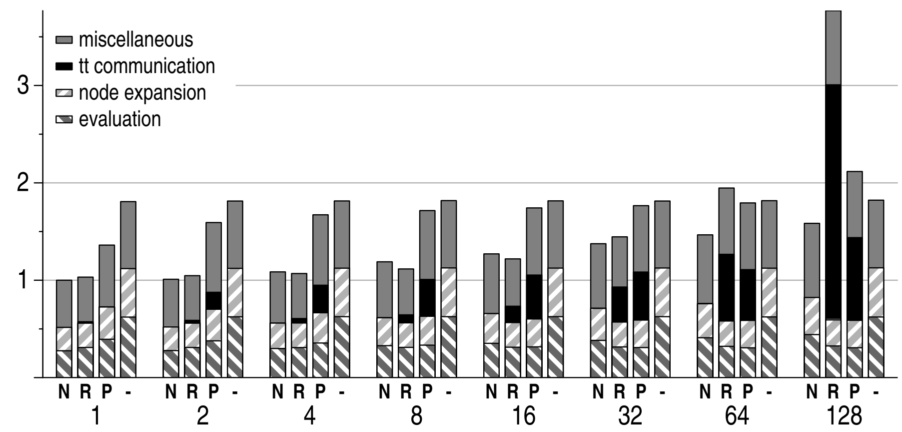


Figure 4.33: 15-puzzle performance breakdown.

The figure shows that much time is spent in “miscellaneous” program parts. This is caused by the low execution times of the move generator and the evaluation function (3.66 and 2.70 μs respectively). Therefore, a relatively large time is spent in the search engine and in the repetition detection module, which together account for the majority of the remaining time.

Non-shared performs best in most cases, except on 4–16 processors. *Replicated* performs better than *partitioned* on up to 32 processors. On 128 processors, *replicated* shows a communication performance dip that is caused by LFC’s flow control mechanism. There is not enough memory on the Myrinet network interfaces to accommodate a reasonable amount of receive buffers,[8] therefore senders quickly run out of send credits and start communicating with the credit manager to get more send credits. These negotiations increase the amount of communication even more and aggravate the problem. The version that does not use a transposition table always searches the same tree, and therefore has the same overhead regardless of the number of processors used.

Our speedups are lower than those published in the literature; for example, Cook and Varnell [34] report 58.90-fold speedups on 64 processors. However, we compare to a much more efficient sequential algorithm that uses the transposition table to reduce redundant search effort.

[8]Current Myrinet cards have 4 or 8 MB, which would suffice for 128 processors; ours are equipped with 1 MB.

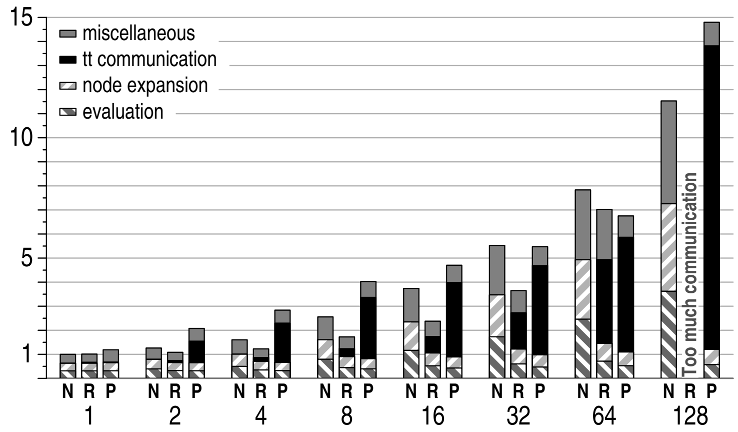


Figure 4.34: Double-blank puzzle performance breakdown.

4.7.5.5 The double-blank puzzle

The double-blank puzzle is a modification to the 15-puzzle, where we removed the "15"-tile, leaving two positions blank. This modification yields a game that builds search trees with many transpositions, because two consecutive moves involving both blanks can usually be interchanged, resulting in the same position. We used this game to stress-test the behavior of the different transposition table implementations.

The double-blank puzzle uses the evaluation function of the 15-puzzle, adapted for two blanks. The test positions are the same positions as those for the 15-puzzle, with the '15'-tile removed. They are shown in Appendix B.3. We limited the search to a 66-ply search depth. *Partitioned* performs transposition table lookups for all nodes in the tree, since the game has so many transpositions. *Replicated* does transposition table updates for all interior nodes. Storing leaf information in the transposition table is expensive and the possible savings are minor.

Figure 4.34 shows why the speedups are at most 11.1, even on 128 processors. The 14-puzzle suffers from an enormous search overhead if the transposition table is not shared, or from an extreme communication overhead if the table is shared. On 128 processors, the search overhead for *non-shared* is a factor of 11.5, and *partitioned* spends 85% of the time doing remote transposition table accesses. We were not able to perform measurements for *replicated* on 128 processors, because LFC collapses when all machines start broadcasting at an uncontrolled rate, despite the flow control.

4.7.5.6 Rubik's cube

The third one-person game used in our test suite is Rubik's cube. We use the move generator that is written in C, since it is faster than the one written in the Multigame

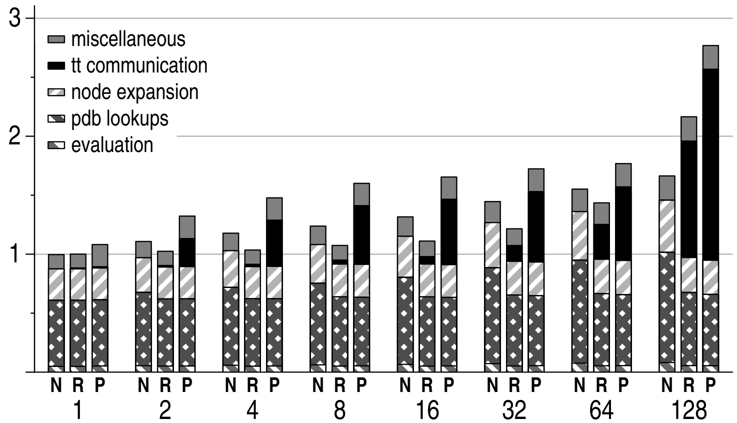


Figure 4.35: Rubik's cube performance breakdown.

language. The evaluation function is very simple, but is assisted by two pattern databases [68], one for the corner pieces and one for the edge pieces. The implementation of the pattern databases is discussed in Section 4.5.4. The pattern databases are read into main memory before the search begins. With respect to the other games, we halved the number of transposition table entries (2^{21} instead of 2^{22}), to leave room for the pattern databases. *Partitioned* performs transposition table lookups for all nodes in the tree, and *replicated* updates all interior nodes.

Rubik's cube was tested using 5 random problems, depicted in Appendix B.1. Since a random problem requires weeks of CPU time to solve, we limited the search depth to 17.

The performance breakdown for Rubik's cube is shown in Figure 4.35. Rubik's cube has many transpositions (two consecutive turns on opposing sides), but these transpositions form small cycles in the search tree. Since Rubik's cube migrates few jobs, these cycles are likely to be searched by a single processor; therefore *non-shared* performs quite well for this application.

On 2 to 64 processors, *replicated* performs best. On 128 processors, LFC's flow control mechanism delays the transposition table communication for both *partitioned* and *replicated*; yet Rubik's cube does not communicate as much as the double-blank puzzle does. The search overheads are minimal.

Contrary to other games, *replicated* performs better than *partitioned* for up to 128 processors. This is due to the wide trees built by Rubik's cube, which implies that there are many leaf nodes. Although the prefetching mechanism of *partitioned* works well for wide trees, *replicated* does not update leaf nodes at all while *partitioned* looks them up as well (at the time of a lookup, the search engine does not yet know whether a node will be leaf or interior). Consequently, the lookup/update ratio for the

game	evaluation	PDB lookup	node expansion
chess	12.7	–	44.0
checkers	141	–	19.1
Othello	76.4	–	91.1
15-puzzle	2.70	–	3.66
double-blank	3.26	–	10.7
Rubik's cube	1.02	11.7	58.9

Table 4.9: Absolute costs of the evaluation functions, pattern database lookups, and node expansions (in μs).

transposition table is high, approximately 11/1.

4.7.6 Discussion and conclusions

We discussed the performance of the Multigame runtime system, using six different applications. We saw that the applications in our test suite exhibit big differences in scalability and the way they spend their times. Yet there are some commonalities, which we will discuss now.

First, for most applications the transposition table is crucial for obtaining good performance. Chess, checkers, Othello, and the double-blank puzzle scale badly if the transposition table is not shared.

Second, *replicated* performs surprisingly well on systems of up to 16 or 32 processors. On larger systems, too much time is spent in the broadcast message handlers, since each invocation consumes a considerable amount of time, and because each broadcast message invokes the handler on *all* machines. Nevertheless, this way of sharing is promising on clusters of SMP machines (such systems are likely to become the most cost-effective parallel machines in the near future), since each SMP needs to process each broadcast message only once, while the other processors in the SMP can continue searching the search tree. Of course, if the processors have more time to search the tree, more communication traffic will be created, and more messages will be broadcast.

Third, for each application there is a minimum number of processors for which *partitioned* starts outperforming *replicated*. On our hardware, this number is usually around 32, although for Rubik's cube *replicated* still outperforms *partitioned* on 128 processors.

The times needed to generate children and to evaluate positions are very different for the tested applications, yet each application spends a considerable amount of time expanding and evaluating nodes. The evaluation function for checkers is slow but carries much knowledge. On the other hand, both the evaluation function and the move generator of the 15-puzzle are fast because they are heavily optimized. Note that the times include the overhead for measuring; approximately 0.16 μs (half the time of a *TimerStart*/*TimerStop*-pair) should be extracted to obtain the real time. The evaluation function of Rubik's cube does little; Rubik's cube mostly depends on the pattern databases.

From the performance breakdown we learned that the applications pass their time throughout the entire program. The general rule-of-thumb that an application spends 90% of the time in 10% of the code does not hold for game playing. The transposition-table broadcast handler for *replicated* is a possible exception; if an application performs updates for (nearly) all nodes in the tree and a large number of processors is used, most time is spent there. However, on a large number of processors it is usually better to use *partitioned* transposition tables. The lack of hot spots in the code makes optimizing a game-playing program a tedious task, since optimizations must be applied everywhere, and each optimization usually saves about 5 or 10% at the application level.

Although we invested much effort optimizing the Multigame software, there is still room for further improvement. For example, we did not implement pattern databases for the 15-puzzle [36, 37], nor did we implement Enhanced Transposition Cutoffs (ETC) [86, 88]. ETC lookups all children of a node before the first child is searched. Search effort is reduced when the first, most promising child does not cause a cutoff, and one of the other children transposes into a state that has been searched previously and does cause a cutoff. Since the number of transposition-table lookups increases significantly, ETC is likely to perform well for *replicated* tables and badly for *partitioned* tables. Moreover, the prefetching mechanism for *partitioned* will not work in that case, because there is no time to overlap communication and computation.

The speedups of each of the six tested games are far from perfect, since tree searching algorithms are hard to parallelize. Three important causes for overhead are *search overhead*, *communication overhead*, and *synchronization overhead*. We observe that the proportions of these overheads differ for each of the tested games. The distributed transposition table contributes significantly to the communication overhead, whether implemented as a *replicated* table or as a *partitioned* table. The communication overhead can be reduced by increasing the threshold for which transposition table accesses are allowed or by not sharing the table at all. Whether this has a major impact on the search overhead depends on whether the game builds trees with many transpositions.

4.8 Experiences with multi-threaded and distributed programming

Implementing a runtime system for distributed game playing turned out to be a hard task, resulting in a major effort to test and debug the software. In this section, we will discuss *why* it is hard to implement a distributed game-playing runtime system.

There are a number of issues that complicate the implementation, some of which are specific for distributed game-tree search. We will, however, first discuss some general problems of programming for distributed memory systems.

First, programming with threads is error-prone. While sequential programs written in an imperative language usually overspecify the execution order, the multi-threaded programming model encourages to write programs that *underspecify* the execution order: after a *ThreadFork()*, any execution order is allowed except if forbidden by synchronization statements, and such synchronization points are easily overlooked.

Second, Panda's upcall model for incoming messages is hard to use. Upcall handlers are not allowed to block and wait for events that are triggered by invocation of other upcall handlers [73]. A second restriction is that a thread must release all mutexes before sending a message. These restrictions are exactly what makes Panda communication fast (a message can always be received on the stack of the currently running thread, eliminating the need for unnecessary thread switches), but from a programmer's point of view, it is sometimes hard to ensure that the rules are not violated. The second restriction implies, for example, that an application cannot atomically enter an item into a message buffer, send the buffer (if entering the item causes the buffer to be full), and clear the buffer for future use.

In current versions of Panda, it is possible to control message receipt by disabling network interrupts and poll the network outside critical sections. This eases programming, because the application can poll for messages outside critical sections, eliminating the need for synchronization. However, in the design phase of the Multigame runtime system we could not rely on controlled message receipt, therefore the runtime system is able to receive messages at all times, provided that the runtime system is compiled with the *multi-threading* option enabled.

Another recent feature of Panda that would have eased the Multigame runtime system implementation, is the ability to FIFO-order messages. The absence of FIFO ordering provides us with many opportunities for potential race conditions. For example, work can be killed even before it arrives. The Multigame runtime system is able to deal with non-FIFO message delivery, but implements this in a rather ad hoc manner. FIFO ordering of messages is only meaningful if network interrupts are disabled, otherwise a message can be received via an interrupt while another message is being received via a poll. Moreover, Panda does not order group and unicast messages relative to each other; the Multigame runtime system deals with this as well.

The lack of suitable parallel debugging tools hampers debugging. We found that most of the time core dumps are created too late to see something useful, because the real error is triggered somewhere else, and often by another thread, or on another machine.

Lock-free programming [54, 59, 117] is hard and error-prone. Lock-free programming techniques synchronize multiple threads by optimistically performing memory operations through the use of atomic compare-and-swap, fetch-and-add, and test-and-set machine instructions. Multiple threads can concurrently perform such an operation, but the code is arranged in such a way that only one thread succeeds in doing the operation, while the concurrent threads have to redo the operation. Programming with locks is much easier, but less efficient because an expensive thread switch is required when a thread wants to acquire a lock that is already taken by another thread. We implemented a fast lock-free memory allocation module for the allocation and deallocation of tree nodes. We had a hard time getting the code correct, because the programming model is so error-prone. It is already hard to avoid race conditions in push and pop operations of a stack, while the stack is one of the "simplest" data structures.

During the development of the Multigame software, we also ran into bugs in the thread scheduler, communication software, operating system, C and C++ compilers, and even suffered from occasional network-memory bit errors. Some of these errors were extremely hard to track down.

There are also issues that make writing a distributed game-playing runtime system hard because of the nature of the application domain.

First, game-tree search is insensitive to many programming errors. We distinguish two kinds of errors which remain easily unnoticed, namely those where the answer is correct but incorrectly computed, and those where the answer is correct but for which too much work has been done. An example of an error in the former category is when the search result of a subtree is ignored due to a race condition; yet the search algorithm is often perfectly able to hide the error and suggest the right move. Errors easily get lost in the tree. Performance errors are even more likely to remain unnoticed. For example, occasional overflows of the history heuristic table counters during deep checkers tree searches drastically disturbed the move ordering. We were not even aware of this error until we carefully analyzed the performance statistics. Another move ordering bug in quiescence search remained unnoticed for over a year. Unfortunately, it is impossible to guarantee program correctness. However, by the careful analysis of the performance measurements, we gained at least some confidence in the correctness of our software.

Second, the non-deterministic nature of game-tree search inherently poses debugging problems. Sequences of actions are seldom repeatable, and each run searches another tree. This makes it often hard to track down a bug.

Third, search trees are hard to visualize. Trees of more than 30 nodes are hard to debug, because one loses the overview over their connection, especially if the state is distributed over several machines. In practice, trees are over 10,000,000 nodes in size, and many bugs are not triggered on small sized trees.

Fourth, the asynchronous nature of different events makes the communication model intricate. For example, Alpha-Beta updates (see Section 4.2.2.6) and kill messages are hard to implement. Apart from the FIFO ordering discussed above, a parent might send a kill message to a child at the same time that the search result for the child is sent to the parent. Therefore, the parent must be able to handle a search result even if it killed the child, and the child must be able to handle a kill message even if the search for it has already finished.

The following debugging strategy proved its usefulness several times. We let the sequential NegaMax search engine search a tree up to a certain depth. The NegaMax search engine searches all nodes in a tree without pruning. For each node and each search depth, we store the search result in a (large) file. To debug a parallel search engine, the search engine reads the entire file into memory before searching the same position. Each time the parallel search engine has computed a lower or upper bound for a node, the search result is matched to the precomputed value read from the file and the program is aborted if the search results are not consistent.

There are higher-level programming models that should ease programming a parallel game-playing runtime system. Cilk [20, 51] probably offers the most suitable model, although the model is too weak to express Alpha-Beta updates (NegaScout and MTD(f) kill messages can be expressed by using the abort mechanism). Moreover, the NegaScout example in the Cilk manual [113] is exceptionally hard to understand. Cilk is discussed extensively in Section 4.3.4.

In conclusion, we postulate that implementing a high performance distributed game-playing runtime system requires significant effort. A low-level programming model, lack of debugging tools, non-determinism, and insensitivity to many kinds of errors make debugging such a runtime system a tedious task.

4.9 Discussion and conclusions

In this chapter, we discussed the Multigame runtime system. The runtime system supports a wide variety of board games on distributed memory systems; shared memory systems are supported as well. The runtime system implements several parallel search algorithms: IDA* for one-player games and Alpha-Beta, NegaScout, and MTD(f) for two-player games. The search engines described in this chapter use random work stealing to distribute work. The runtime system has two interfaces to the move generator: one that supports code generated by the Multigame front-end compiler, and one that is easier for use by humans.

Writing a distributed runtime system for a game-playing environment is a difficult task. Especially if the underlying message passing software does not support FIFO ordering or if message delivery is interrupt driven (i.e., can occur at any time), race conditions must be prevented everywhere. Polled message delivery is easier to program with, and is more efficient for fine-grained communication. The nature of the application domain also complicates an implementation. The non-deterministic behavior of parallel search, the fact that many errors remain unnoticed, and the large (distributed) state space make debugging a tedious task.

The runtime system implements a number of game-independent heuristics to guide the search. The transposition table is used for preventing searching the same subtree multiple times, and for move-ordering. Three distribution mechanisms for the transposition table were implemented: replicated, partitioned, and non-shared tables. The history heuristic is also used for move-ordering. The history tables are occasionally synchronized during distributed searches. Quiescence search extends the search at positions where the evaluation value cannot be trusted due to material or tactical disturbance. Cycles in the directed search graph are efficiently detected, and pattern databases considerably reduce the search space in one-player games.

The runtime system is highly optimized. Since a game-playing program tends to spend the time everywhere in the program code, many optimizations are needed that each have a modest impact on application performance. Speculatively started work in progress that becomes obsolete is killed by update messages. Memory management for dynamic node allocation is efficient, even in multi-threaded environments. The transposition-table communication is reduced using several techniques. We found that application-specific network interface firmware reduces remote lookup latencies considerably; unfortunately network interfaces are hard to program. Prefetching also helps in reducing remote lookup latencies. We implemented game-specific optimizations as well. Some optimizations have not yet been implemented, such as Enhanced Transposition Cutoffs, and 15-puzzle pattern databases.

We did extensive performance measurements for six different Multigame applications, using various transposition-table distribution schemes. Each of the applications was tuned for best performance. Parameter tuning turned out to be a tedious task; automating this process would relieve the programmer from this burden. Moreover, tuning had to be done for each separate number of processors. Speedups are modest due to search overhead, communication overhead, and synchronization overhead. Modest speedups are common for parallel search algorithms, and our speedups compare well to other work. The impact of each of the overheads differed for the various applications. Despite the transposition table communication optimizations, setting thresholds for remote lookup or update operations is still necessary, although the table can be shared to levels near the leaves. A surprising result is the good performance for replicated transposition tables on up to 32 processors.

The programmer has to select some components from the runtime system and tune some parameters to obtain good parallel performance. In this respect, we did not completely succeed in hiding parallelism. Nevertheless, the programmer requires almost no knowledge about parallelism, and this model is much easier to use than to write a parallel game-playing program from scratch.

Chapter 5

Transposition-table-driven work scheduling in distributed search

In the previous chapter, we described the Multigame runtime system. We analyzed the performance for several Multigame applications, showing that transposition-table accesses are one of the primary sources of overhead. We experimented with partitioned, replicated, and local transposition tables, and concluded that all approaches have serious scaling problems. Partitioned-table lookups are expensive since they are blocking. Replicated tables communicate too much data when many processors are used, since each table update is broadcast to, and handled by all other processors. Using local tables results in significantly higher search overheads.

These table distribution schemes are intuitive ways to implement a distributed transposition table. However, we believe that the traditional way to implement distributed search, using *work stealing*, disallows an efficient implementation of a distributed transposition table. Without a transposition table, work stealing is efficient (since work stealing itself involves little communication overhead). But if one first parallelizes the search algorithm and subsequently adds a distributed transposition table as an afterthought, it is hard to get a table entry to the place where it is needed: at the processor that processes the corresponding state.

In this chapter, we investigate a different approach to parallelize a search algorithm, and integrate the scheduling of the parallel search algorithm with the transposition table. The key idea is to partition the transposition table over the processors, and to *migrate* a state to the processor that owns the corresponding table entry, rather than using work stealing. We call this approach transposition-table-driven work scheduling, or *Transposition-Driven Scheduling* (TDS) for short. At first, this idea seems counterintuitive, since it requires much communication. However, since all table accesses are local now, we get rid of the blocking remote table lookups, avoiding processor idle times. Moreover, the asynchronous communication patterns allow latency hiding, and message combining can even further improve the communication performance.

We implemented TDS for the IDA* [67] search algorithm, which resulted in a new search engine. This search engine was embedded in the Multigame runtime system, as shown in Figure 4.1. Here, our TDS implementation is an instance of the box labelled “search engine”, and replaces the cluster with the boxes “job queue”, “work stealing”, and “path encoding”. So far, we implemented TDS only for single-agent search, since straightforward implementation of two-agent search is more difficult due to back-propagation of local search results.

In this chapter, we report on our experiences with TDS for IDA*. The chapter is organized as follows. In Section 5.1, we describe the TDS algorithm, and how it works for IDA*. Then, in Section 5.2, we discuss some implementation issues. Section 5.3 lists the advantages and disadvantages of TDS. Section 5.4 reports and discusses performance results for several applications. Although we did not implement TDS for two-player algorithms, we discuss some two-player related issues in Section 5.5. Section 5.6 describes related work, and in Section 5.7, we draw conclusions.

Results of this chapter were published in [100].

5.1 The basic algorithm

TDS is a distributed scheduling algorithm, and, like work stealing, is built on top of a search algorithm. The scheduling algorithm describes where and when states are expanded. Work stealing naturally clusters subtrees on individual processors, but TDS scatters the tree over all processors. At first sight, this seems illogical, since TDS communicates much more than work stealing does. However, distributed transposition tables are hard to implement efficiently when combined with work-stealing based scheduling algorithms. TDS avoids the problem by integrating the scheduling and the transposition table, lowering both communication and search overheads.

Figure 5.1 illustrates how TDS for IDA* works; the numbers in this paragraph correspond to the black numbers in the figure. Each processor stores part of the transposition table (1), and has a local work queue (2). The local work queue contains states that need expansion. As long as there are states in the work queue, the processor takes a job, and expands it to its successor states (3). After expansion, the parent state is destroyed. Each child is evaluated, using an admissible evaluation function. States that are too far from a target (i.e., the evaluation function returns a minimum distance that is greater than the state’s search depth) are pruned (4). Each of the remaining states is hashed to a transposition table entry, and sent to the processor that owns the entry (5). Upon arrival, the state is looked up in the transposition table. If the state is not there (6), the entry is both written into the transposition table and into the local job queue (7). If the state is already in the table (8), the state is a transposition,

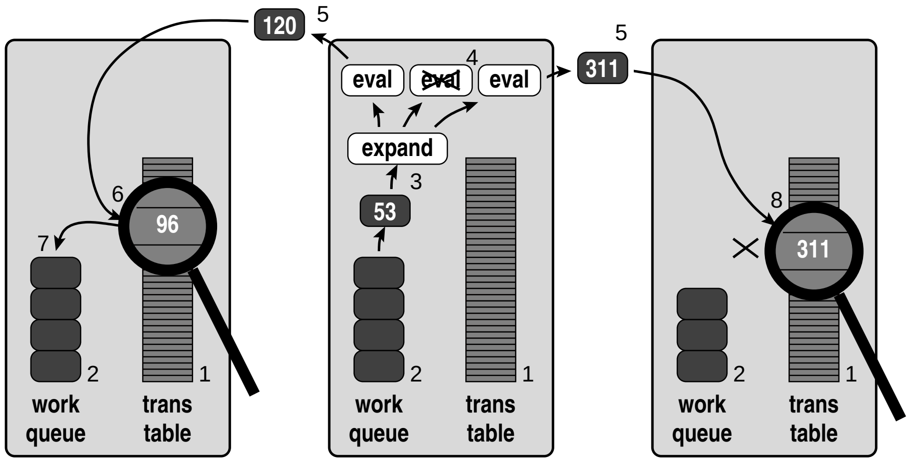


Figure 5.1: Transposition-Driven Scheduling for IDA*. Black numbers are referred to in the text.

and need not be searched again.

Each state is assigned a *home processor*, which manages the transposition table entry for this state. The home processor is computed from the state's signature. Some of the signature bits indicate the processor number of the state's home, while some of the remaining bits are used as an index into the transposition table at that processor.

Figure 5.2 shows the pseudo code for a Transposition-Driven Scheduling algorithm, which is executed by every processor. The function *MainLoop* repeatedly tries to retrieve a state from its local work queue. If the queue is not empty, it expands the state on the head of the queue by generating the children. Then it checks for each child whether the lower bound on the solution length (obtained by *Evaluate*) exceeds the IDA* search bound, in which case it causes a cutoff. If not, the child is sent to its home processor. When the local work queue is empty, the algorithm checks whether all other processors have finished their work and no work messages are in transit. If there is still work somewhere, it waits for new work to arrive.

The function *ReceiveState* is invoked for each state that is received by a processor. The function first does a transposition table lookup to see whether the state has been searched before. If not, or if the state has been searched to an inadequate depth (e.g., by a previous iteration of IDA*), the state is stored into the transposition table and put into the local work queue; otherwise the state is discarded because it has transposed into a state that has already been searched adequately.

The values stored in the transposition table are used differently for work stealing and TDS. With work stealing, a table entry stores a *search result* (a lower bound on the minimal distance to the target), derived by searching the subtree below it. Finding a transposition table entry with a suitably high table value indicates that the state has

```
PROCEDURE MainLoop()
   WHILE NOT Finished DO
      State := GetLocalJob();
      IF State <> NULL THEN
         Children := ExpandState(State);
         FOR EACH Child IN Children DO
            IF Evaluate(Child) <= Child.SearchBound THEN
               Dest = HomeProcessor(Signature(Child));
               SendState(Child, Dest);
            END
         END
      ELSE
         Finished := CheckGlobalTermination();
      END
   END
END


PROCEDURE ReceiveState(State)
   Entry := TransLookup(State);
   IF NOT Entry.Hit OR Entry.SearchBound < State.SearchBound THEN
      TransStore(State);
      PutLocalJob(State);
   END
END
```

Figure 5.2: Simplified TDS algorithm.

been previously searched adequately. With TDS, an entry contains a *search bound*. It indicates that the subtree below the state has either been previously searched adequately (as above) or is currently being searched with the given bound. Note that this latter point represents a major improvement over previous distributed transposition table mechanisms in that it prevents two processors from ever working on the same subtree concurrently.

5.2 Implementation issues

We now discuss some implementation issues of this basic algorithm. An important property in our TDS implementation of IDA* is that a child state does not report its search result to its parent. As soon as a state has forked off new work for its children, work on the state itself has completed. In some cases (for example, for two-person, minimax search algorithms) the results of a child should be propagated to its parent. Applicability of TDS to two-person games is discussed in Section 5.5.

Since no results are propagated to the parent, the TDS algorithm needs a separate mechanism to detect global termination. TDS synchronizes after each IDA* iteration, and starts a new iteration if the current iteration did not solve the problem. One of the many distributed termination detection algorithms can be used. We use the time count algorithm described in [77], which counts the size of the local work queues and the number of pieces of work in transit. The overhead for termination detection is negligible, because new iterations are started infrequently, and because the termination detection algorithm is active only when a work queue becomes empty.

Another issue concerns the search order. Scheduling prescribes not only on which processor a state is expanded, but also in which order. It is desirable to do the parallel search in a depth-first way as much as possible, because breadth-first search will quickly exhaust the memory for intermediate states. Depth-first behavior could be achieved using priority queues, by giving work on the left-hand side of the search tree a higher priority than that on the right-hand side of the tree. However, manipulating priority queues is expensive. Instead, we implement each local work queue as a stack, at the possible expense of a larger working set. When searching sequentially, a stack corresponds to pure depth-first search.

An interesting trade-off concerns when and where to invoke the evaluation function. One option is to do the evaluation on the processor that creates a piece of work, and to migrate the work to its home processor only if the evaluation did not cause a cutoff. Another option is to migrate the work immediately to its home processor, look it up in the transposition table, and then call the evaluation function only if the lookup did not cause a cutoff. The first approach will migrate less work but will always invoke the evaluation function, even if the state has been searched before (on the home processor). Which approach is more efficient depends on the relative costs for migrating and evaluating states. On our system, the first approach performs best for most applications.

An important optimization performed by our implementation is message combining. To decrease the overhead per migrated state, several states that have the same source and the same destination processors are combined into one physical message. Each processor maintains a message buffer for every other processor. A message buffer is transmitted when it is full, or when the sending processor has no work to do; this typically happens during the start and the end of each iteration, when there is little work.

The best results for TDS are achieved when all processors get an equal amount of work on average. If some processors are slower than others, or if the load is not evenly balanced, then the stacks of work of the processors that cannot keep up will grow progressively. Priority queues will not help in this case, since they will not keep fast processors from generating work too fast. A flow control scheme would be required to slow down processors that send states too frequently. We have not found the need to implement such a mechanism yet.

5.3 Discussion

Transposition-Driven Scheduling has six advantages:

1. All transposition table accesses are local.

2. All communication is asynchronous; processors do not wait for messages. As a result, the algorithm scales well to large numbers of processors. The total bandwidth requirements increase approximately linearly with the number of processors.

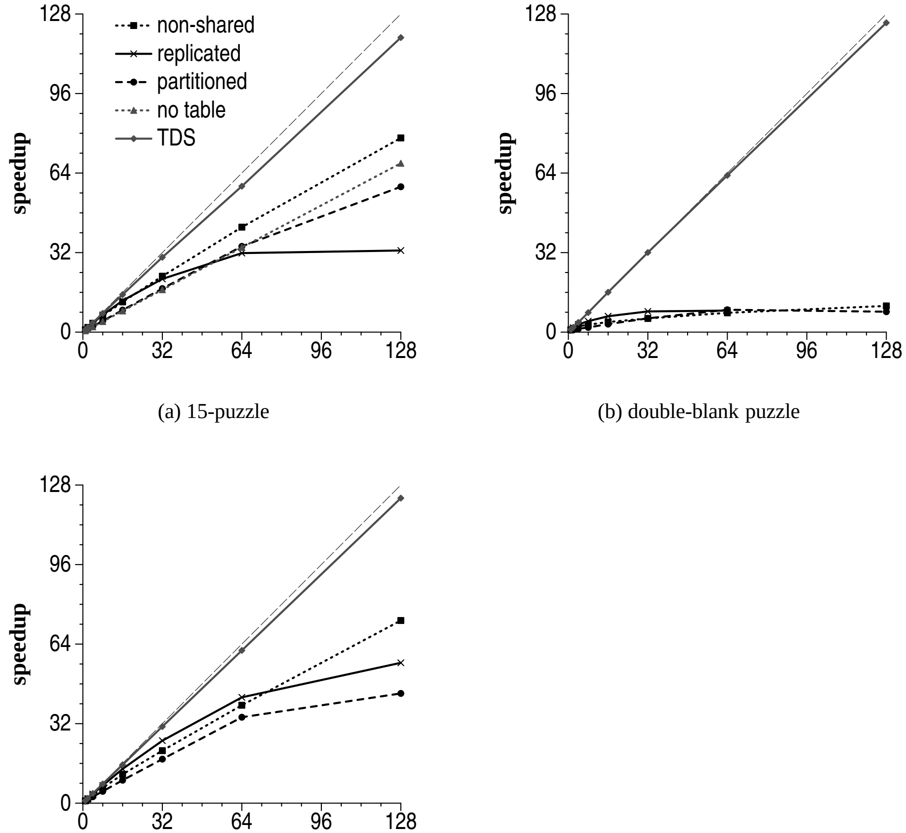
3. No duplicate searches are performed. With work stealing, multiple processors may concurrently search a transposition because the transposition-table update occurs *after* the subtree below it was searched. With the new scheme this cannot occur; all attempts to search a given subtree must go through the same home processor. Since it has a record of all completed and in-progress work in the transposition table, it will not allow redundant effort. The only situation in which duplicate work will get done, is when the transposition table is too small for the given search. Some table entries will get overwritten, and this loss of information can result in previously completed searches being repeated.

4. TDS uses the extra memory that comes when more processors are added in an efficient way. The extra memory is used to cache more states during long searches, which decreases the likelihood that entries are evicted from the table.

5. TDS produces more stable execution times for trees with many transpositions than the work-stealing algorithm.

6. No separate load-balancing scheme is needed. Previous algorithms require work stealing or some other mechanism to balance the work load. Load balancing in TDS is done implicitly, using the hash function. Most hash functions, including the one we use [122], are uniformly distributed, causing the load to be distributed evenly over the machines. This works well as long as all processors are of the same speed. If this is not the case, the stacks of the slow processors will grow and may exhaust memory. A flow control scheme can be added to keep processors from sending states too frequently. In our experiments, we have not found the need to implement such a mechanism.

We determine the root's search bound of a new IDA* iteration as follows. During an iteration we compute for each node that is pruned the difference between its evaluation value and its search bound. Each processor maintains the local minimum of the differences seen so far. If an iteration does not lead to a solution, the next iteration will be started with a search bound that is increased by the global minimum of the differences. Determining the global minimum hardly requires extra communication, since the local minima can be collected during global termination detection.



(a) 15-puzzle

(b) double-blank puzzle

(c) Rubik's cube

Figure 5.3: Average application speedups.

5.4 Performance measurements

We compare the performance of TDS with that of work stealing, with *partitioned*, *replicated*, and *non-shared* transposition tables. Our test suite consists of three games: the *15-puzzle*, the *double-blank puzzle* (see Section 4.7.5.5), and *Rubik's cube*. Since the 15-puzzle has few transpositions, we include numbers for a variant that uses no transposition table at all. The conditions under which we measure the performance for *TDS* are the same as those for the other variants. The performance of the other variants is described in more detail in Section 4.7. Unlike *partitioned*, *TDS* does not use customized firmware (see Section 4.4.4) but runs on top of LFC and Panda.

Figure 5.3 shows speedups with respect to *TDS* searching on a single processor, the fastest variant for sequential searches for all applications. *TDS* outperforms *partitioned* and *replicated* by a factor 2.0 to 15.1 on 128 processors; *non-shared* is outperformed by a factor 1.5 to 11.8. *TDS* scales almost linearly.

Figure 5.4 shows the performance breakdown for the applications. The results are visualized the same way as in Chapter 4. The height of each bar reflects how much time is spent in a particular program part. The differences in heights explain the overheads. The bars labeled "N" represent the results for *non-shared*, "R" for *replicated*, "P" for *partitioned*, "–" for *no table*, and "T" for *TDS*. For this variant, the black areas represent the time to communicate the work to other processors, rather than the time to communicate remote transposition-table entries. The graphs are normalized; the norm is a sequential *TDS* run, which is the fastest sequential variant for all applications. More information on the construction of the bar graphs can be found in Section 4.7.5.

On a single processor, we see that *TDS* spends more time to evaluate states. This is due to our decision to evaluate a state on the processor where it is created, rather than the one to which it is sent. Therefore, a state is evaluated before it is looked up in the transposition table, even if the state is a transposition and needs no re-evaluation. It is possible to optimize the local case where a state is created on the processor that contains the corresponding transposition table entry, and perform the transposition table lookup before evaluation takes place. This would increase the performance of *TDS* on small-scale systems even more.[1]

The gray shaded areas (which represent the time spent in the remaining program parts) for *TDS* are smaller than for the other variants. Due to its simplicity, the search engine of *TDS* is considerably faster than the other search engines. Moreover, *TDS* does not require the *position repetition detection* module (see Section 4.5.3) to detect cycles in the directed search graph. *TDS* detects repetition of positions through the transposition table, because *TDS* updates the transposition table *before* a state is searched.

Figure 5.4 also illustrates that *TDS* performs well on large-scale systems. The increase in transposition table size and the involved decrease in search effort largely compensates the increase in communication overhead. Load imbalance turned out to be negligible; the most busy processor does typically less than 1% more work than the least busy processor.

Even on 128 processors, *TDS* uses only a small fraction of the available Myrinet bandwidth, which is about 60 MByte/s per link between user processes. The 15-puzzle requires 2.2 MByte/s, the double-blank puzzle 1.4 MByte/s, and Rubik's cube 0.36 MByte/s. Each job is encoded in 32–68 bytes. For all games we combine up to 64 pieces of work into one message. The communication overhead for distributed termination detection (*TDS* synchronizes after each iteration) is well below 0.1% of

[1] As a consequence, the self-relative speedups for *TDS* would decrease, but the *TDS*-relative speedups for the other variants would lag behind even more.

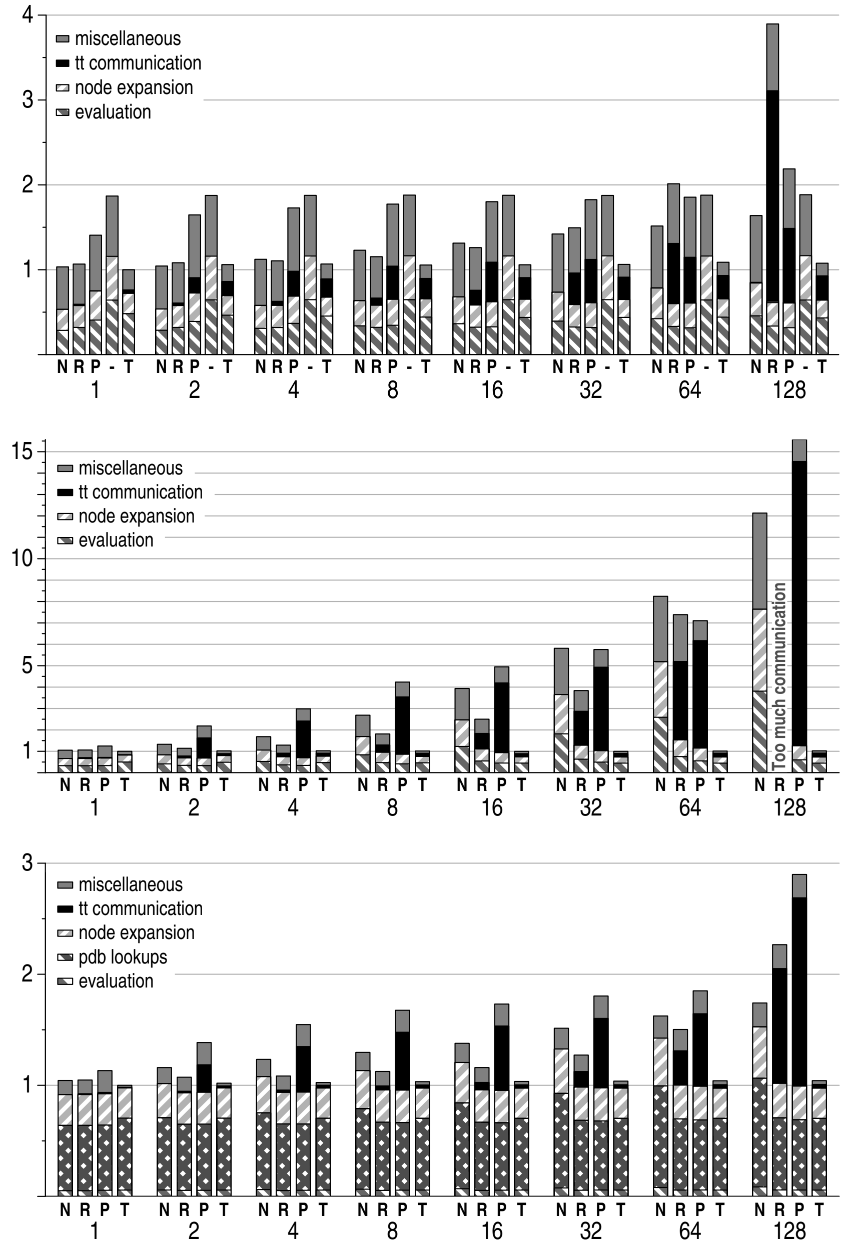


Figure 5.4: Performance breakdown for the 15-puzzle (top), the double-blank puzzle (middle), and Rubik's cube (bottom).

the total communication overhead.

Partitioned suffers from high lookup latencies. Even with the customized network firmware, a remote lookup takes typically 10–25 μs, including the overhead for prefetching (see Section 4.4.5). For the double-blank puzzle on 128 processors we even measured an average lookup latency of 58 μs; here the flow control mechanism in the network software slows down the application to prevent it from overrunning receive buffers. At a rate of 14,400 remote lookups per second per processor, the application spends 85% of the time communicating table entries.

Like *TDS*, *partitioned* profits from the increase in table size when more processors are added. Yet the performance graph for the double-blank puzzle, which has many transpositions, shows an 84% search overhead on 128 processors. We explain this as follows. *Partitioned* (as well as *replicated* and *non-shared*) updates the transposition table *after* the search of a state completes. A transposition is not recognized as such before the update is performed, thus *partitioned* may search a transposition multiple times by multiple processors to the same depth concurrently. This phenomenon does not occur with *TDS*, since the table update is done *before* the state is searched.

Replicated passes most of its time handling incoming broadcast messages when many processors are used. We were not able to perform measurements for the double-blank puzzle on 128 processors, because LFC cannot handle the communication load when all machines broadcast data too frequently. The other measurements on many processors show a significant communication overhead.

The speedups of *TDS* through 64-processors for the 15-puzzle are similar to those reported by others (e.g., [34] reports 58.90-fold speedups). However, previous work has only looked at parallelizing the basic IDA* algorithm, usually using the 15-puzzle with Manhattan distance as the test domain. The state of the art has progressed significantly. For the 15-puzzle, the linear conflicts heuristic [58] reduces tree size by roughly a factor of 10; transposition tables reduce tree size by an additional factor of 1.8 (and more for larger table sizes); and the last move and corner conflict heuristics [69] reduce the tree size even more. These reductions result in a less well balanced search tree, increasing the difficulty of achieving good parallel performance. Still, our performance is comparable to the results in [34]. This is a strong result, given that the search trees are tens of times smaller.

5.5 Applicability to two-person search algorithms

We have shown that TDS is particularly suitable to parallelize the one-person IDA* search algorithm. The next challenge would be to apply TDS to two-person search algorithms like Alpha-Beta, MTD(f), and NegaScout. More research is needed to evaluate the applicability of TDS to this class of search algorithms, but we list a number of issues that potentially complicate TDS for two-person search algorithms.

Two-person search algorithms are minimax algorithms. This implies that the value of a parent depends on the search results of its children. Unlike TDS for IDA*, each

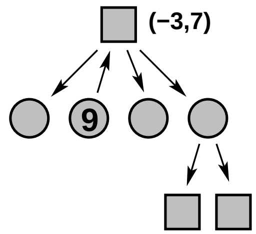


Figure 5.5: Pruning in TDS for two-player search algorithms.

search result has to be reported to its parent. TDS for IDA* stores *search bounds* in the transposition table, and updates entries *before* a state is searched. Two-player search algorithms require *search results*, which can only be stored *after* the state is searched. This is a fundamental difference with TDS for IDA*, although this does not necessarily imply that TDS is unsuitable for two-player search algorithms. A consequence of updating the transposition table *after* the tree is searched, is that multiple processors may search the same subtree at the same moment. This phenomenon does not occur with TDS for IDA*.

Another issue concerns pruning (or Alpha-Beta window narrowing, which was discussed in Section 4.2.2.6), as illustrated by Figure 5.5. Assume that the current root window is (3,7), and one of the children returns the result 9. The other children can then be pruned. However, the children, and possibly their children as well, are already spread over the other processors, or somewhere on their way. Since each child (and the children's children) is likely to be stored on another processor, many messages may be needed to kill all children.

The search order in two-person search algorithms is important. The leaf states should be visited in left-to-right order as much as possible. TDS for IDA* used a stack to implement the local work queues. The stack turned out to preserve the depth-first search order of IDA* reasonably well, but it remains to be seen whether it will not disturb the left-to-right search order too much.

Message combining in two-person search algorithms cannot be applied as gratuitously as in IDA*. The parent updates should not be queued too long, since they may cause a cutoff (or a narrowed search window) at the parent. Even if it does not cause a direct cutoff at the parent, it might cause a cutoff at one of the grandparents if the child happens to be the last child that reports the value to its parent.

TDS for two-person search algorithms will not achieve the almost linearly increasing speedups achieved for IDA*. TDS will not solve the search overhead caused by speculatively searching tree nodes (see Section 4.2.2.3).

From the above, we conclude that TDS cannot be applied to two-player search algorithms without further research. Nevertheless, there is reason to believe that TDS for two-player algorithms will work. The communication characteristics will probably improve due to latency hiding, since TDS communicates asynchronously. A comparison between TDS and work stealing based algorithms would be interesting.

5.6 Related work

Numerous parallel single-agent search algorithms have appeared in the literature. The most popular ones are task distribution schemes that partition the search tree over the available processors [92]. Task distribution can be simplified by expanding the tree in a breadth-first fashion until the number of states on the search frontier matches the number of processors [70]. This can cause load balancing problems (the search effort required for a state varies widely), implying that enhancements, such as work stealing, are necessary for high performance. A different approach is Parallel Window Search (PWS) [89], where each processor is given a different IDA* search bound for its search. All processors search the same tree, albeit to different depths. Some processors may search the tree with a search bound that is too high. Since sequential IDA* stops searching after using the right search bound, PWS results in much wasted work. Asynchronous IDA* (AIDA*) [96] uses a combination of a data partitioning scheme and work stealing, and allows processors to search to different depths concurrently.

All these schemes essentially considered only the basic IDA* algorithm, without important search algorithm enhancements that significantly reduce the search tree size (such as transposition tables).

IDA* uses less space than A*. This comes at the expense of expanding additional states. The simple formulation of IDA* does not include the detection of duplicate states (such as a cycle, or transposing into a state reached by a different sequence of state transitions). The transposition table is a convenient mechanism for using space to solve these search inefficiencies, both in single-agent [94] and two-player [110] search algorithms. There are other methods, such as finite state machines [115], but they tend to be not as generally applicable or as powerful as transposition tables.

By integrating transposition table access with work scheduling, TDS makes all communication asynchronous, allowing communication and computation to overlap. Much other research has been done on overlapping communication and computation [42]. The idea of self-scheduling work dates back to research on data flow and has been studied by several other researchers (see, for a discussion, [38]). In the field of problem solving, there are some cases in which this idea has been applied successfully. In software verification, the parallel version of the Murphi protocol verifier uses its hash function to schedule the work [112]. In game playing, a parallel generator of end-game databases (based on retrograde analysis) uses the Gödel numbers of states to schedule work [8]. In single agent search, a parallel version of A*, PRA*, partitions its OPEN and CLOSED lists based on the state [47].

Interestingly, the papers present the data-flow-like parallelization as following in a natural way from the problem at hand, and, although the authors report good speedups, they do not compare their approaches to more traditional parallelizations. The paper on PRA*, for example, does discuss differences with IDA* parallelizations, but focuses on a comparison of the *number* of state expansions, without addressing the benefit of asynchronous communication for *run times*.[2] (A factor may be that PRA* was designed for the CM-2, a SIMD machine whose architecture makes a direct comparison with recent work on parallel search difficult.)

Despite the good performance of data-flow-like parallelization, so far no in-depth performance study between work stealing and data-flow-like approaches such as TDS has been performed for distributed search algorithms.

5.7 Conclusions

Efficient parallelization of search algorithms that use transposition tables is a challenging task, due to communication overhead and search overhead. We have described a new approach, called Transposition-Driven Scheduling (TDS), which integrates work scheduling with the transposition table. TDS makes all communication asynchronous, overlaps communication with computation, and reduces search overhead.

We implemented parallel IDA* using TDS, and performed a detailed comparison of TDS to the conventional work stealing approach on a large-scale parallel system. TDS performs significantly better, especially for large numbers of processors. On 128 processors, TDS achieves a speedup between 109 and 122, where traditional work-stealing algorithms achieve speedups between 8.7 and 62. TDS scales well to large numbers of processors, because it effectively reduces both search overhead *and* communication overhead.

TDS represents a shift in the way one views a search algorithm. The traditional view of single-agent search is that IDA* is at the heart of the implementation, and performance enhancements, such as a transposition tables, are added in afterwards. This approach makes it hard to achieve good parallel performance when one wants to compare to the best known sequential algorithm. With TDS, the transposition table becomes the heart of the algorithm, and performance improves significantly.

[2] Evett et al. compare PRA* against versions of IDA* that lack a transposition table. Compared to IDA* versions with a transposition table, PRA*'s node counts would have been less favorable.

Chapter 6

Conclusions and discussion

In this thesis we presented an environment for distributed game-tree search, called Multigame. The environment, implements the following research objectives:

- to provide the application programmer with a programming model that is easy to use and avoids explicit parallelism; and

- to provide the researcher in the field with an experimental environment for research on distributed game-tree search.

The first objective is addressed by offering the application programmer a very high-level language, in which the rules of a board game are expressed. The language is easy to use and its expressiveness allows most games to be implemented easily. The Multigame front-end compiler generates a move generator from the rules of a game. To play well, the programmer should provide an evaluation function in C as well, using a simple interface to the Multigame runtime system. The move generator, the evaluation function, and the components in the Multigame runtime system compose a program that plays the game in parallel on a distributed (or shared) memory system.

The second objective is achieved by the design and implementation of a modular and extendable runtime system for distributed game-tree search. The infrastructure provides the game-tree researcher with an experimental environment in which new search techniques and heuristics can be empirically tested using a variety of applications.

The runtime system is highly optimized. We observe that the parallelism is exploited exclusively by the runtime system, not by the front-end compiler. This is a logical choice, since the front-end compiler creates move generators in which the parallelism is too fine grained to be efficiently exploited on off-the-shelf hardware. The parallelism is exploited and controlled by the search engines, although many runtime system components have dedicated support for execution on distributed platforms.

6.1 The Multigame language

Chapter 3 discussed the Multigame language. A Multigame program describes the rules of a board game in a formal way. The language offers the application programmer a simple programming model, based on a combination of the Logo and Prolog programming paradigms. Most board games can be easily expressed in the Multigame language. The rules of most games can be described in a program of just some tens of lines of program code.

We implemented a prototype front-end compiler that creates a move generator from a Multigame program. The generated code uses backtracking to find the legal moves from a given board position. The compiler emits C code, which is portable and efficient. We have found, however, that two omissions in the C language complicate the implementation, namely the absence of inter-procedural gotos and the impossibility to take the address of an arbitrary statement and use it as a first-class object (i.e., one cannot assign such an address to a variable of the appropriate type). The result is an intricate scheme that is a little less efficient than what would have been obtainable when these two constructs were allowed.

A programmer is not obliged to describe the rules of a game in the Multigame language; a hand-written move generator in C can be used instead. There are two reasons why a programmer would prefer a hand-written move generator: the game might have a rule that is hard to express in the Multigame language, or the programmer is not satisfied with the performance of the move generator generated by the front-end compiler. An alternative interface to the runtime system assists the C programmer, because this interface is easier to use than the default interface used by the front-end compiler. We have implemented hand-written move generators for the 15-puzzle, chess, and Rubik's cube.

For most games the front-end compiler generates reasonably efficient code. Usually a hand-written move generator is some tens of percents faster than an equivalent move generator generated by the front-end compiler. For chess, however, our hand-written move generator is 9.6 times as fast as the front-end compiler generated move generator; this is due to the chess rule that the king may not be left in check, which requires an expensive test.

Another reason for performance loss is the choice for the target language C. We have also implemented an experimental front-end compiler that generates IA-32 assembly instead of C. The experimental compiler does not implement the entire Multigame language, but can be used for a few simple games. Performance results show that the experimental front-end compiler generates faster code; for the 15-puzzle the generated code is just as fast as a hand-written move generator in C.

6.2 The runtime system

Chapter 4 discussed issues related to the Multigame runtime system. The runtime system provides game-independent functionality like search engines and search heuristics. The runtime system is optimized for efficient game-tree search on a distributed system. Since game-playing programs spend the time in many program parts rather than in an isolated piece of code, we had to optimize many components to obtain a substantial overall improvement.

The extensibility and flexibility of the runtime system allows the game-tree researcher to experiment with new search techniques and optimizations. We used the runtime system to develop a new scheduling technique for parallel IDA* search, which is described in Chapter 5 and is summarized below.

We also used the runtime system to experiment with different transposition-table distribution techniques: *partitioned*, *replicated*, and *non-shared*. Non-shared tables only work well for applications with few transpositions or with transpositions that involve few moves, so that a transposition is likely to be encountered multiple times by the same processor. Replicated transposition tables work surprisingly well on small and medium-scale systems, provided that enough network bandwidth is available. On large-scale systems, the broadcast message handlers form a bottleneck. Partitioned tables become larger as more processors are added, but suffer from synchronous remote lookup latencies.

Remote transposition lookup latencies are determined largely by the roundtrip communication time with the server machine, which is dominated by the time that the receiving processor needs to react to an incoming request message. Interrupting the processor is too expensive, since it involves several kernel-user space context switches. The alternative, having the CPU poll the network frequently for incoming request messages, increases the lookup latency, since the CPU has other work to do and cannot poll at all times. We modified the firmware on the network interface processors to handle an incoming lookup request message *itself* immediately after arrival, rather than dispatching the message to the CPU (see Section 4.4.4). Programming network interface processors is hard, since programming errors usually crash the entire machine and few debugging tools exist. The performance is increased significantly; measurements for several applications show that the lookup latencies are reduced by 44% to 70%. Application level performance increases by 31% to 47% when lookups are performed at all depths in the tree. Even better performance could be obtained by writing new network processor firmware from scratch, rather than extending existing firmware (LFC). This would, however, require significantly more programming effort.

We devised and implemented a prefetching scheme that reduces processor idle times due to remote lookup latencies even more. An asynchronous lookup request message is sent to retrieve transposition table information for a node that is likely to be searched in the near future. This will reduce or completely eliminate processor idle time when the node is actually searched. Performance experiments for a number of applications show that prefetching usually does not hide the latency completely, but is nevertheless beneficial.

The runtime system contains other game-independent and game-dependent heuristics as well. An example of a game-independent heuristic is the history heuristic, which is used to order the game tree. The history table is weakly synchronized on a distributed system. Game-dependent heuristics can be implemented in an evaluation function. The runtime system also supports pattern databases.

Section 4.7 shows extensive performance results for various Multigame applications. For each of the applications we show how search overhead, communication overhead, and synchronization overhead contribute to imperfect speedups. This is done using several transposition table implementations, and we illustrate the efficiencies for the different sharing strategies under varying circumstances. We show that no single strategy outperforms another under all circumstances, and that the performance differences can be large.

We found that implementing a runtime system for distributed game playing is hard (see Section 4.8). Programming with threads or message passing is error-prone. Race conditions easily occur, especially when message ordering is not guaranteed or when message receipt is interrupt driven. Lock-free programming is dangerous, even when applied to simple data structures. The non-deterministic nature of distributed game-tree search renders debugging a tedious task. Moreover, programming errors remain easily unnoticed, because a search result is often correct, even when it is established incorrectly.

6.3 Transposition-Driven Scheduling

Chapter 5 presents a new scheduling algorithm for distributed IDA*, which we call Transposition-Driven Scheduling. The algorithm combines the work distribution with a distributed transposition table. Traditional approaches are based on work stealing. This is hard to combine with a distributed transposition table, because synchronous remote lookups must be performed if the transposition table is partitioned, and large amounts of table updates must be broadcast if the table is replicated. TDS eagerly migrates a state to the processor where the corresponding transposition table entry is stored. At first this seems expensive, because nearly all states have to be communicated, but message combining reduces the communication overhead, and the asynchronous nature of the communication reduces processor idle times. We compared the performance of TDS to traditional work-stealing approaches with different transposition table distribution techniques for several applications, and show that TDS outperforms work stealing by a wide margin, especially on large clusters. Traditional implementations achieve efficiencies between 6.4% and 61% on 128 processors; TDS achieves speedups that are close to perfect.

Since TDS is asynchronous, we presume that the algorithm is insensitive to large latencies. Future work should give insight into whether TDS, combined with the necessary optimizations to reduce the bandwidth requirements, is suitable for execution on wide-area distributed systems (metacomputers). Another direction for future research is to apply TDS to two-player search algorithms. Since this kind of (minimax) algorithms requires propagation of search results upward in the tree, it is not yet clear whether TDS will be successful for this class of algorithms.

6.4 Did we reach our goals ?

Programming a distributed memory machine for fast, parallel game-tree search is a hard task. We felt that building an integrated environment would involve many scientific challenges. As stated in the introduction, we had two objectives in mind: provide the application programmer with a programming model that is easy to use and avoids explicit parallelism; and provide the researcher in the field with an experimental environment to do research on distributed game-tree search. Did we reach these goals ? The answers are: *mostly*, and *yes*.

Since the second answer is the simpler one, we will first elaborate on it. The Multigame environment is a very suitable tool for game-tree research. This is demonstrated by the fact that we were able to implement a new parallel search algorithm (TDS) in just a few days, thereby showing that the runtime system provides the right infrastructure to experiment with new algorithms. We also mention the research on distributed transposition tables (network firmware experiments, prefetching) and the research on the Multigame language. The variety of applications can be used to validate the usefulness of a new idea.

The first question has a more complicated answer. We *did* provide the programmer with an easy programming model; writing a distributed game-playing program is much easier using the Multigame environment than writing one from scratch. The Multigame language is free from explicit parallel statements, yet the programmer has to be aware that the program runs on a distributed system and that it communicates. The programmer does have to choose the right transposition-table distribution technique, the right lookup and update thresholds, and the right depth to which parallelism is allowed, in order to obtain the best performance.

Ideally, timing the configuration parameters should be done automatically; currently the optimal settings must be found manually. Tuning the system for optimal performance is tedious and time consuming, because many test positions must be searched to get an accurate impression of the performance of the system under a particular set of configuration parameters. An environment that tunes the system automatically would be useful. Several techniques are feasible; the most promising is a self-monitoring runtime system that changes behavior whenever it feels that the current settings are suboptimal. The advantage of a dynamic system is that it adapts to the changing shapes of the game trees in the course of a game. This is important, since many games build trees that have different shapes in the opening, middle, and end phase.

Another limitation is that the Multigame system is unable to derive an evaluation function automatically from the rules of a game. We consider this too far beyond the current state-of-the-art in Artificial Intelligence, and to be a research topic on its own. To play well, the programmer has to write an evaluation function in C. Since the evaluation function is sequential, this does not put a real burden upon the programmer. A high-level language to describe evaluation functions may be helpful for the programmer.

Nevertheless, the Multigame environment assists the application programmer in such a way that board games can be implemented and run on distributed memory systems without requiring the programmer to have extensive knowledge about parallelism. New games can often be implemented in a matter of hours; implementing a simple evaluation function adds another few hours or days. This makes the Multigame environment particularly useful for prototyping newly developed games as well. During the course of this research we implemented about 25 games, some of which were devised recently.

Appendix A

Example Multigame programs

A.1 The 15-puzzle

```
players =  1
dimensions (4,4)

pieces
{
    piece_1         '1'
    piece_2         '2'
    piece_3         '3'
    piece_4         '4'
    piece_5         '5'
    piece_6         '6'
    piece_7         '7'
    piece_8         '8'
    piece_9         '9'
    piece_10        'A'
    piece_11        'B'
    piece_12        'C'
    piece_13        'D'
    piece_14        'E'
    piece_15        'F'
}

main =              move_piece,
                    try goal_reached.

move_piece =        find empty field,
                    orthogonal,
                    step,
                    pick up,
                    step backward,
```

```
                    put down.

goal_reached =      match (empty field, piece_1 , piece_2 , piece_3 ,
                           piece_4,     piece_5 , piece_6 , piece_7 ,
                           piece_8,     piece_9 , piece_10, piece_11,
                           piece_12,    piece_13, piece_14, piece_15),
                    win.
```

A.2 Connect-4

```
dimensions (7,6)
sides = 1

pieces
{
    mark            'X' 'O'
}

main =              try new_mark else draw.

new_mark =          irreversible,
                    find empty field,
                    not [ south, step, points at empty field ],
                    replace by mark,
                    try winning_position.

winning_position =  any direction,
                    optionally [ step, points at own piece ],
                    repeat 3 times
                        [ step backward, points at own piece ],
                    win.
```

A.3 Chess

```
# This implementation does not check the 50-move rule !

dimensions (8,8)

pieces
{
    pawn            'P' 'p'
    rook            'R' 'r'
    knight          'N' 'n'
    bishop          'B' 'b'
    queen           'Q' 'q'
    king            'K' 'k'
```

```
}

properties
{
    may_castle_left, may_castle_right        : player : [ 0 .. 1 ]

    en_passant_column                        : board  : [ 0 .. 8 ]
    # 0      – no en passant pawn
    # 1 .. 8 – pawn at (en_passant_column,4) may be captured en passant
}


main =              try legal_move else try check_mate else draw.

legal_move =        either pawn_move or rook_move or knight_move or
                           bishop_move or queen_move or king_move or
                           castle_move,
                    try [
                        not find opponent's king,
                        win,
                    ] else test [
                        find own king,
                        not attacked,
                    ].

pawn_move =         irreversible,
                    find own pawn,
                    pick up,
                    either pawn_forward_move or pawn_capture_move,
                    put down,
                    try promotion.

pawn_forward_move = north,
                    step,
                    points at empty field,
                    either [                # double forward
                        assert (row == 3),
                        step,
                        points at empty field,
                        assign (en_passant_column = 9 – column),
                    ] or assign (en_passant_column = 0).

pawn_capture_move = either northwest or northeast,
                    step,
                    either points at opponent's piece or [
                        # en passant move
                        points at empty field,
                        assert (en_passant_column == column),
                        south,

                        step,
```

```
                            points at opponent’s pawn,
                            replace by empty field,
                            step backward,
                        ],
                        assign (en_passant_column = 0).

promotion =             assert (row == 8),
                        either replace by queen or replace by knight or
                               replace by bishop or replace by rook.

rook_move =             find own rook,
                        try [
                            assert (row == 1 && column == 1 &&
                                    may_castle_left),
                            irreversible,
                            assign (may_castle_left = 0),
                        ] else try [
                            assert (row == 1 && column == 8 &&
                                     may_castle_right),
                            irreversible,
                            assign (may_castle_right = 0),
                        ],
                        pick up,
                        orthogonal,
                        repeat 0 .. infinity times [
                            step,
                            points at empty field,
                        ],
                        step,
                        not points at own piece,
                        put down,
                        assign (en_passant_column = 0).

bishop_move =           find own bishop,
                        pick up,
                        diagonal,
                        repeat 0 .. infinity times [
                            step,
                            points at empty field,
                        ],
                        step,
                        not points at own piece,
                        put down,
                        assign (en_passant_column = 0).

queen_move =            find own queen,
                        pick up,
                        any direction,
                        repeat 0 .. infinity times [
                            step,
                            points at empty field,
```

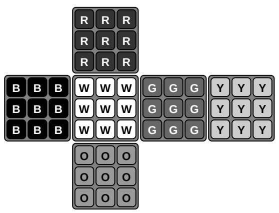
```
                    ],

                    step,
                    not points at own piece,
                    put down,
                    assign (en_passant_column = 0).

king_move =         find own king,
                    pick up,
                    any direction,
                    step,
                    not points at own piece,
                    put down,
                    try [
                        assert(may_castle_left || may_castle_right),
                        irreversible,
                        assign(may_castle_left = may_castle_right = 0),
                    ],
                    assign (en_passant_column = 0).

knight_move =       find own knight,
                    pick up,
                    orthogonal,
                    step,
                    either turn 45 or turn −45,
                    step,
                    not points at own piece,
                    put down,
                    assign (en_passant_column = 0).

castle_move =       irreversible,
                    try [
                        white to move,
                        move to (5,1),
                    ] else move to (4,1),
                    either [
                        assert (may_castle_left),
                        west,
                    ] or [
                       assert (may_castle_right),
                       east,
                    ],
                    test [
                        step,
                        do [
                            points at empty field,
                            step,
                        ] while not points at own rook
                    ],
                    test repeat 3 times [ not attacked, step ],
                    assign (may_castle_left = may_castle_right = 0),
```

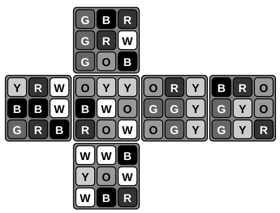
```
                    pick up,
                    repeat 2 times step,

                    put down,
                    while step do nothing,
                    pick up,
                    do step backward while not points at own king,
                    step backward,
                    put down,
                    assign (en_passant_column = 0).

attacked =          test either atckd_by_pawn or atckd_by_rook or
                                atckd_by_knight or atckd_by_bishop or
                                atckd_by_queen or atckd_by_king.

atckd_by_pawn =     either northwest or northeast,
                    step,
                    points at opponent's pawn.

atckd_by_rook =     orthogonal,
                    do step while points at empty field,
                    points at opponent's rook.

atckd_by_bishop =   diagonal,
                    do step while points at empty field,
                    points at opponent's bishop.

atckd_by_queen =    any direction,
                    do step while points at empty field,
                    points at opponent's queen.

atckd_by_king =     any direction,
                    step,
                    points at opponent's king.

atckd_by_knight =   orthogonal,
                    step,
                    either turn 45 or turn -45,
                    step,
                    points at opponent's knight.
```

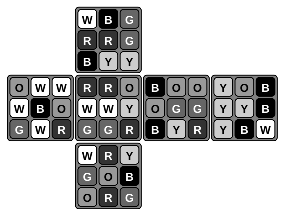
Appendix B

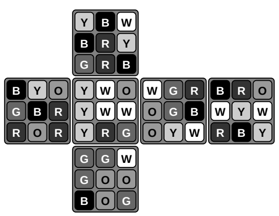
Test positions used

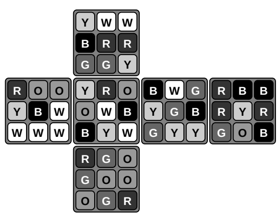
B.1 Rubik's cube

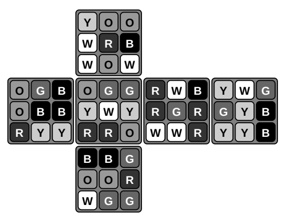

target


01


02


03


04


05

B.2 15-puzzle

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

target

15	14	8	12
10	11	9	13
2	6	5	1
3	7	4	

01

15	11	13	12
14	10	8	9
7	2	5	1
3	6	4	

02

15	11	13	12
14	10	8	9
2	6	5	1
3	7	4	

03

15	11	9	12
14	10	13	8
6	7	5	1
3	2	4	

04

15	11	9	12
14	10	13	8
2	6	5	1
3	7	4	

05

15	11	8	12
14	10	13	9
2	7	5	1
3	6	4	

06

15	11	9	12
14	10	8	13
6	2	5	1
3	7	4	

07

15	11	8	12
14	10	9	13
2	6	5	1
3	7	4	

08

15	11	8	12
14	10	9	13
2	6	4	5
3	7	1	

09

B.3 Double-blank puzzle

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

target

	14	8	12
10	11	9	13
2	6	5	1
3	7	4	

01

	11	13	12
14	10	8	9
7	2	5	1
3	6	4	

02

	11	13	12
14	10	8	9
2	6	5	1
3	7	4	

03

	11	9	12
14	10	13	8
6	7	5	1
3	2	4	

04

	11	9	12
14	10	13	8
2	6	5	1
3	7	4	

05

	11	8	12
14	10	13	9
2	7	5	1
3	6	4	

06

	11	9	12
14	10	8	13
6	2	5	1
3	7	4	

07

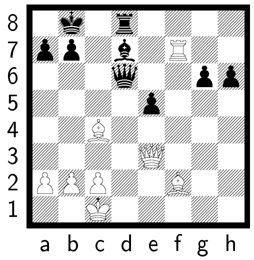
	11	8	12
14	10	9	13
2	6	5	1
3	7	4	

08

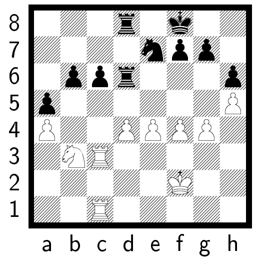
	11	8	12
14	10	9	13
2	6	4	5
3	7	1	

09

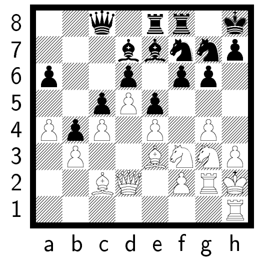
B.4 Chess



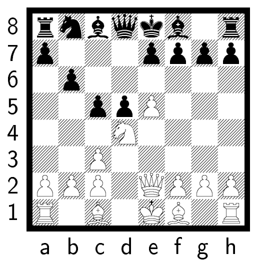
01, black to move



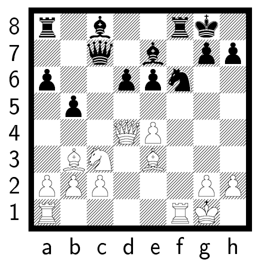
02, white to move



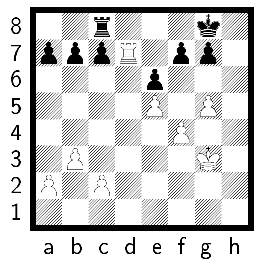
03, black to move



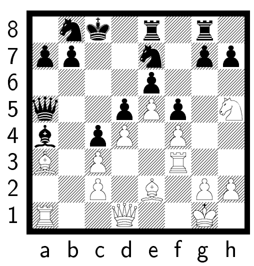
04, white to move



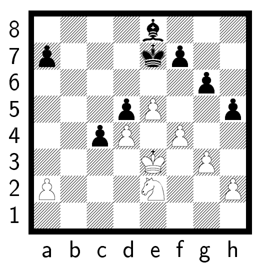
05, white to move



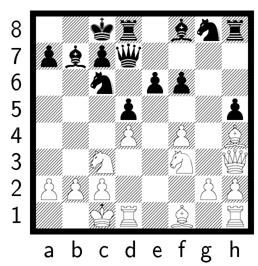
06, white to move



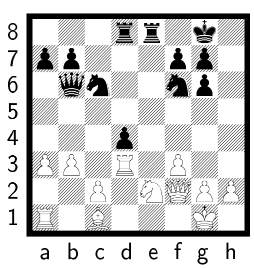
07, white to move



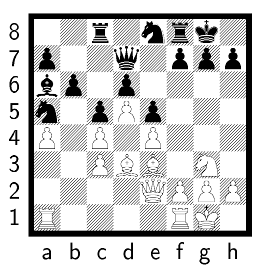
08, white to move



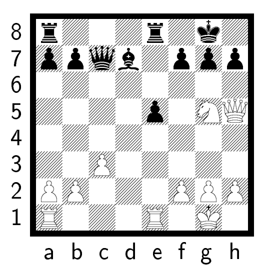
09, white to move



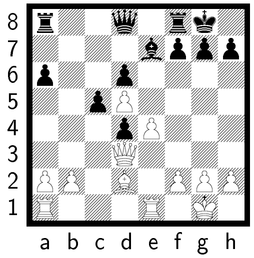
10, black to move

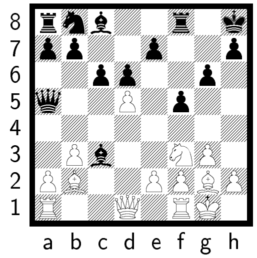


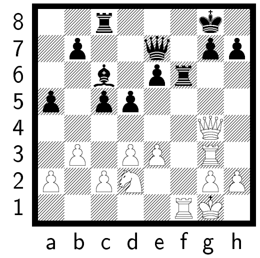
11, white to move

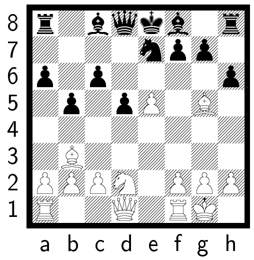


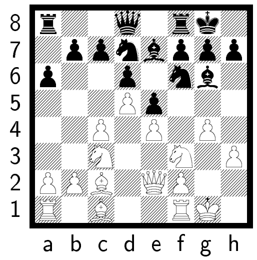
12, black to move

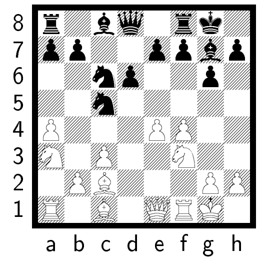

13, white to move

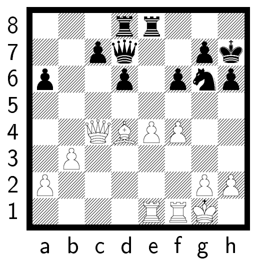

14, white to move

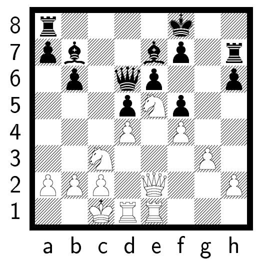

15, white to move

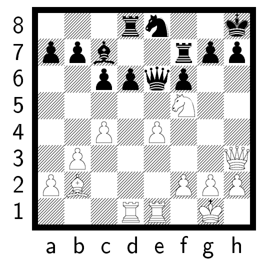

16, white to move

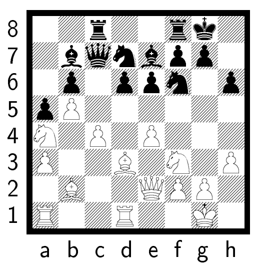

17, black to move

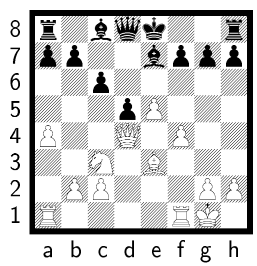

18, black to move

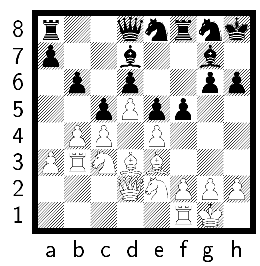

19, black to move


20, white to move

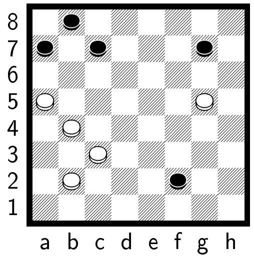

21, white to move


22, black to move

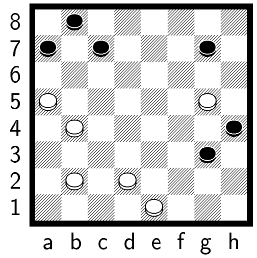

23, black to move


24, white to move

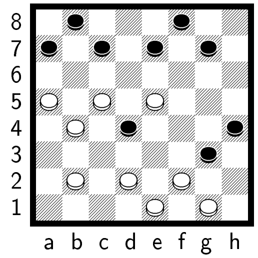
B.5 Checkers



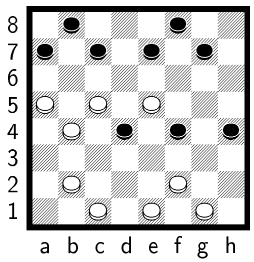
01, black to move



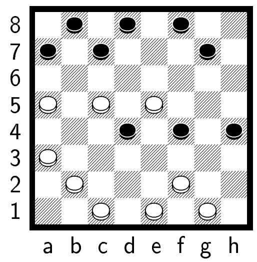
02, black to move



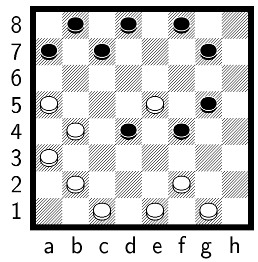
03, black to move



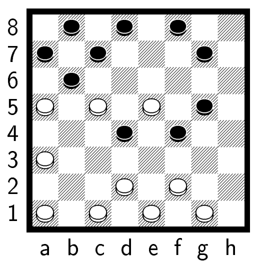
04, black to move



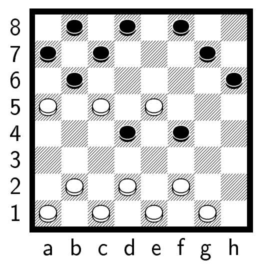
05, black to move



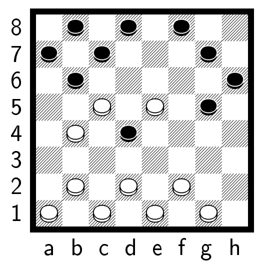
06, black to move



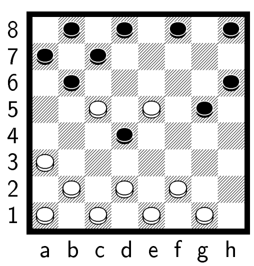
07, black to move



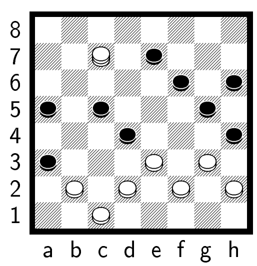
08, black to move



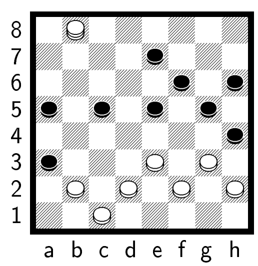
09, black to move



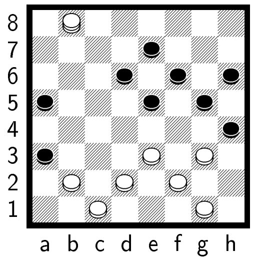
10, black to move



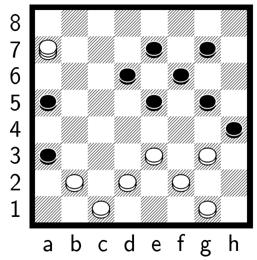
11, white to move



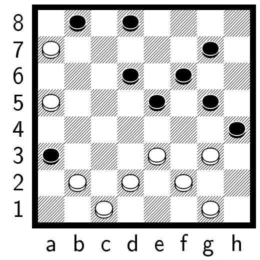
12, white to move



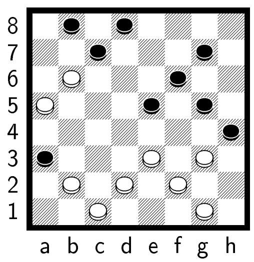
13, white to move



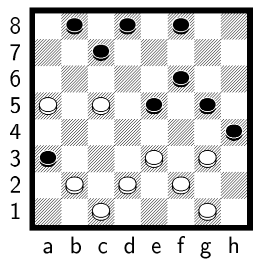
14, white to move



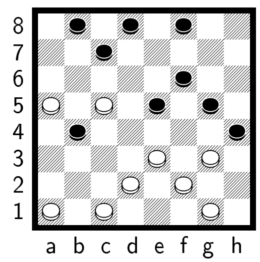
15, white to move



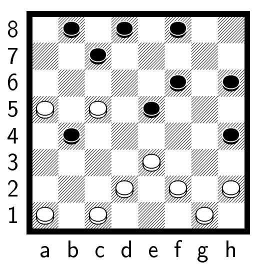
16, white to move



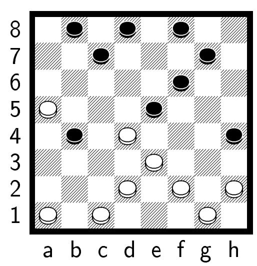
17, white to move



18, white to move

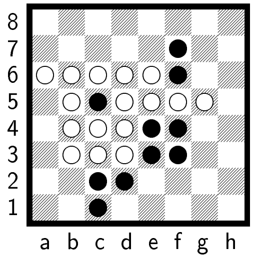


19, white to move

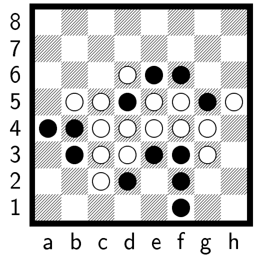


20, white to move

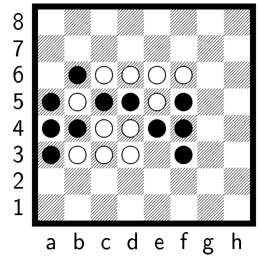
B.6 Othello



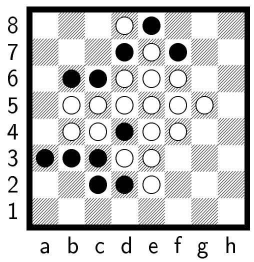
01, black to move



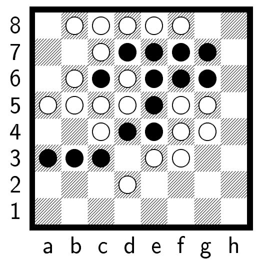
02, white to move



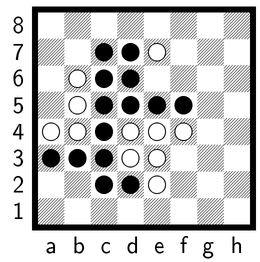
03, black to move



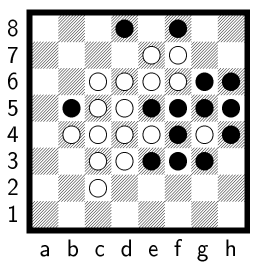
04, white to move



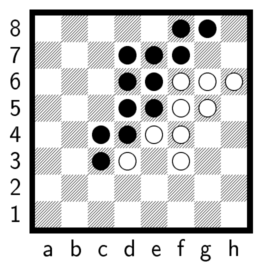
05, black to move



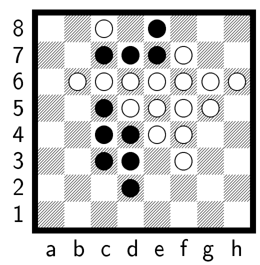
06, white to move



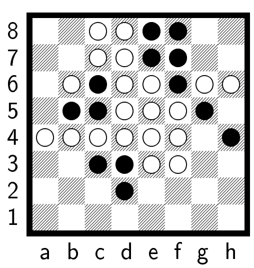
07, black to move



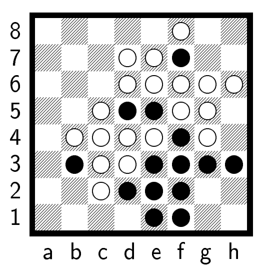
08, white to move



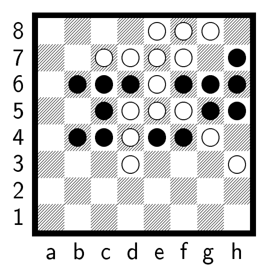
09, black to move



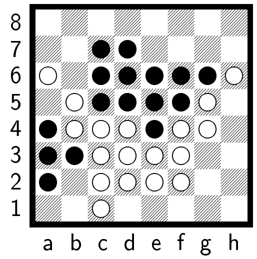
10, white to move



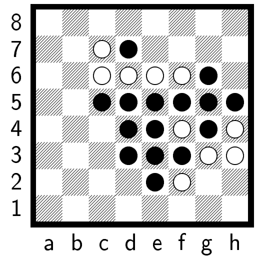
11, black to move



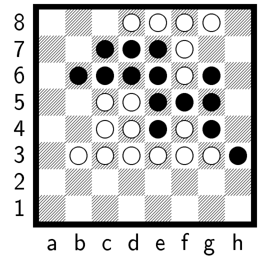
12, white to move



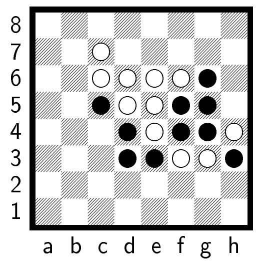
13, black to move



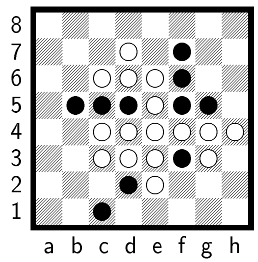
14, white to move



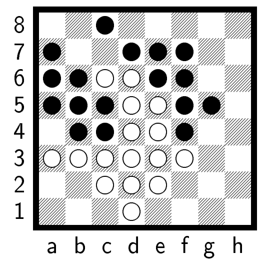
15, black to move



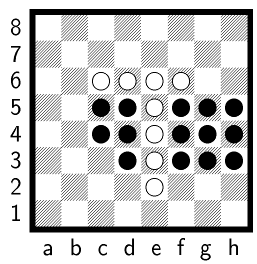
16, white to move



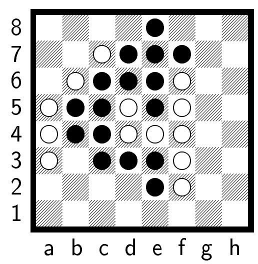
17, black to move



18, white to move



19, black to move



20, white to move

Appendix C

User interface commands

command	*description*
configuration	show program configuration
depth [*n*]	print / set search depth
increment [*n*]	print / set iterative deepening increment value
echo *str*	print *str*
evaluate	print evaluation value of current position
execute *file*	execute *file* with Multigame commands
help [*command*]	show help text (on *command*)
hint	search tree and print best move
legal	show list of legal moves
move	search tree and do best move
move *mov*	force move *mov*
opening *mov*	add *mov* as opening move
play [***white*** | ***black*** | ***off***]	computer is white/black player; automatically search tree and do move if computer is to make a move
pv	show principal variation
quit | *<Crtl-D>*	quit program
random	do a random move
read [*file*]	read position (from *file*)
reset	clear heuristics tables and reset statistics counters
selfplay	play a full game
shell [*command*]	start Bourne shell (and execute *command*)
statistics [*id-list*] [*cpu-list*]	print statistics (about *id-list*) (on CPUs *cpu-list*)

Continued on next page

Continued from previous page	
`time` [*sec*]	print / set time per move *(not yet implemented)*
`undo`	undo move
`verbose` [*level*]	print / set verbosity level
`write` [*file*]	print current position (to *file*)
command $<$ *file*	*command* gets input from *file*
command $>$ *file*	*command* writes output to *file*
command $>>$ *file*	*command* appends output to *file*

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

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

[3] L.V. Allis, M. van der Meulen, and H.J. van den Herik. Proof-Number Search. *Artificial Intelligence*, 66(1):91–124, 1994.

[4] E. Altmann, T.A. Marsland, and T. Breitkreutz. Accounting for Parallel Tree Search Overheads. In *International Conference on Parallel Processing*, volume III, Algorithms and Applications, pages 198–201, University Park, Penn., August 1988.

[5] T.S. Anantharaman, M.S. Campbell, and F.H. Hsu. Singular Extensions: Adding Selectivity to Brute-Force Searching. *Artificial Intelligence*, 43(1):99–109, 1990.

[6] G.R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[7] D.I. August, D.A. Connors, S.A. Mahlke, J.W. Sias, K.M. Crozier, B.-C. Cheng, P.R. Eaton, Q.B. Olaniran, and W. m.W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *International Symposium on Computer Architecture*, pages 227–237, July 1998.

[8] H.E. Bal and L.V. Allis. Parallel Retrograde Analysis on a Distributed System. In *Supercomputing '95*, San Diego, CA, December 1995.

[9] H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M.F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, February 1998.

[10] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[11] D.F. Beal. Experiments with the Null-Move. In D.F. Beal, editor, *Advances in Computer Chess 5*, pages 65–79. Elsevier Science Publishers, Amsterdam, 1989.

[12] D.F. Beal. A Generalized Quiescence Search Algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.

[13] D.F. Beal and M.C. Smith. Multiple Probes of Transposition Tables. *Journal of the International Computer Chess Association*, 19(4):227–233, 1995.

[14] H.J. Berliner. Some Necessary Conditions for a Master Chess Program. In *International Joint Conference on Artificial Intelligence*, pages 77–85, Stanford, MA, 1973.

[15] R.A.F. Bhoedjang. *Communication Architectures for Parallel-Programming Systems*. PhD thesis, Vrije Universiteit, Amsterdam, the Netherlands, June 2000.

[16] R.A.F. Bhoedjang, J.W. Romein, and H.E. Bal. Optimizing Distributed Data Structures Using Application-Specific Network Interface Software. In *International Conference on Parallel Processing*, pages 485–492, Minneapolis, MN, August 1998.

[17] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. Efficient Multicast On Myrinet Using Link-Level Flow Control. In *International Conference on Parallel Processing*, pages 381–390, Minneapolis, MN, August 1998.

[18] R.A.F. 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.

[19] R.D. Blumofe, M. Frigo, C.F. Joerg, C.E. Leiserson, and K.H. Randall. Dag-Consistent Distributed Shared Memory. In *International Parallel Processing Symposium*, pages 132–141, Honolulu, Hawaii, April 1996.

[20] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *ACM SIGPLAN Symposium on Principles and Practice on Parallel Programming*, pages 207–216, Santa Barbara, CA, July 1995.

[21] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[22] I. Bratko and D. Kopec. A Test for Comparison of Human and Computer Performance. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 31–56. Pergamon Press, Oxford, 1982.

[23] D.M. Breuker. *Memory versus Search in Games*. PhD thesis, Universiteit Maastricht, the Netherlands, 1998.

[24] D.M. Breuker, H.J. van den Herik, J.W.H.M Uiterwijk, and L.V. Allis. A Solution for the GHI Problem for Best-First Search. In H.J. van den Herik and Hiroyuki Iida, editors, *Computers and Games (LNCS 1558)*, pages 25–49, November 1998.

[25] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement Schemes and Two-Level Tables. *Journal of the International Computer Chess Association*, 19(3):175–180, 1996.

[26] M.G. Brockington. A Taxonomy Of Parallel Game-Tree Search Algorithms. *Journal of the International Computer Chess Association*, 19(3):162–174, September 1996.

[27] M.G. Brockington. *Asynchronous Parallel Game-Tree Search*. PhD thesis, University of Alberta, Edmonton, Alberta, Canada, November 1997.

[28] M.G. Brockington. Computer Chess meets Planning. *Journal of the International Computer Games Association*, 23(2):85–93, June 2000.

[29] M.G. Brockington and J. Schaeffer. The APHID Parallel Alpha-Beta Search Algorithm. In *Symposium on Parallel and Distributed Processing*, pages 432–436, New Orleans, October 1996.

[30] M.G. Brockington and J. Schaeffer. APHID Game-Tree Search. In H.J. van den Herik and J. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 69–91. Universiteit Maastricht, the Netherlands, 1997.

[31] M. Buro. *Techniken für die Bewertung von Spielsituationen anhand von Beispielen*. PhD thesis, Universität–GH–Paderborn, Paderborn, Germany, December 1994. In German.

[32] M. Buro. The Othello Match of the Year: Takeshi Murakami vs. Logistello. *Journal of the International Computer Chess Association*, 20(3):189–193, 1997.

[33] G. Buzzard, D. Jacobson, M. MacKey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-managed Interface Architecture. In *Operating System Design and Implementation*, pages 245–259, Seattle, WA, October 1996.

[34] D. Cook and R. Varnell. Maximizing the Benefits of Parallel Search Using Machine Learning. In *AAAI National Conference*, pages 559–564, July 1997.

[35] J. Culberson and J. Schaeffer. Efficiently Searching the 15-Puzzle. Technical Report 94-08, Department of Computing Science, University of Alberta, 1994.

[36] J. Culberson and J. Schaeffer. Searching with Pattern Databases. In *Advances in Artificial Intelligence, CSCSI (LNAI 1081)*, pages 402–416. Springer-Verlag, May 1996.

[37] J. Culberson and J. Schaeffer. Pattern Databases. *Computational Intelligence*, 14(4):318–334, 1998.

[38] D.E. Culler, K.E. Schauser, and T. von Eicken. Two Fundamental Limits on Dataflow Multiprocessing. In *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, FL, January 1993. North-Holland.

[39] T.L. Dean and M. Boddy. An Analysis of Time-Dependent Planning. In *AAAI National Conference*, pages 49–54, August 1988.

[40] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects '97*, Stanford, CA, April 1997.

[41] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Symposium on Operating System Principles*, pages 303–316, Copper Mountain, CO, December 1995.

[42] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schausser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[43] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[44] R.A. van Engelen. *CTADEL: A Generator of Efficient Numerical Codes*. PhD thesis, Rijksuniversiteit Leiden, the Netherlands, October 1998.

[45] S.L. Epstein. Prior Knowledge Strengthens Learning to Control Search in Weak Theory Domains. *International Journal of Intelligent Systems*, 7:547–586, 1992.

[46] S.L. Epstein, J. Gelfand, and J. Lesniak. Pattern-Based Learning and Spatially-Oriented Concept Formation with a Multi-Agent, Decision-Making Expert. *Computational Intelligence*, 12(1):199–221, 1996.

[47] M. Evett, J. Hendler, A. Mahanti, and D. Nau. PRA*: Massively Parallel Heuristic Search. *Journal of Parallel and Distributed Computing*, 25:133–143, 1995.

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

[49] R. Feldmann, P. Mysliwietz, and B. Monien. Studying Overheads in Massively Parallel MIN/MAX-Tree Evaluation. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 94–103, 1994.

[50] R.A. Finkel and U. Manber. DIB — A Distributed Implementation of Backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[51] M. Frigo, C.E. Leiserson, and K.H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[52] R. Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. PhD thesis, ETH Zürich, Switzerland, 1995.

[53] R.D. Greenblatt, D.E. Eastlake III, and S.D. Crocker. The Greenblatt Chess Program. In *Proceedings of the Fall Joint Computing Conference*, pages 801–810, San Fransisco, 1967.

[54] M.G. Greenwald and D. Cheriton. The Synergy Between Non-blocking Synchronization and Operating System Structure. In *Operating System Design and Implementation*, pages 123–136, Seattle, WA, October 1996.

[55] D. Grune and C.J.H. Jacobs. A Programmer-friendly LL(1) Parser Generator. *Software — Practice and Experience*, 18(1):29–38, January 1988.

[56] F. h. Hsu. IBM's Deep Blue Chess Grandmaster Chips. *IEEE Micro*, 19(2):70–81, April 1999.

[57] S. Hamilton and L. Garber. Deep Blue's Hardware-Software Synergy. *IEEE Computer*, 30(10):29–35, 1997.

[58] O. Hansson, A. Mayer, and M. Yung. Criticizing Solutions to Relaxed Models Yields Powerful Admissible Heuristics. *Information Sciences*, 63(3):207–227, 1992.

[59] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15:745–770, 1993.

[60] R.C. Holte and I.T. Hernádvölgyi. A Space-Time Tradeoff for Memory-Based Heuristics. In *AAAI National Conference*, pages 704–709, July 1999.

[61] IA-64 Application Developer's Architecture Guide, May 1999. `http://developer.intel.com/design/ia64/downloads/adag.htm`.

[62] C.F. Joerg and B.C. Kuszmaul. Massively Parallel Chess. In *Third DIMACS Parallel Implementation Challenge*, October 1994.

[63] A. Junghanns and J. Schaeffer. Search Versus Knowledge in Game-Playing Programs Revisited. In *International Joint Conference on Artificial Intelligence*, pages 692–697, 1997.

[64] A. Junghanns, J. Schaeffer, M. Brockington, Y. Bjornsson, and T.A. Marsland. Diminishing Returns for Additional Search in Chess. In H.J. van den Herik and J. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 53–67. Universiteit Maastricht, the Netherlands, 1997.

[65] A. Kierulf, K. Chen, and J. Nievergelt. Smart Game Board and Go Explorer: A Study in Software and Knowledge Engineering. *Communications of the ACM*, 33(2):152–166, 1990.

[66] D.E. Knuth and R.W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[67] R.E. Korf. Depth-first Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.

[68] R.E. Korf. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *AAAI National Conference*, pages 700–705, July 1997.

[69] R.E. Korf and L.A. Taylor. Finding Optimal Solutions to the Twenty-Four Puzzle. In *AAAI National Conference*, pages 1202–1207, August 1996.

[70] V. Kumar and V. Rao. Scalable Parallel Formulations of Depth-first Search. In V. Kumar, P. Gopalakrishnan, and L. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 1–42. Springer-Verlag, 1990.

[71] B.C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1994.

[72] B.C. Kuszmaul. The StarTech Massively Parallel Chess Program. *Journal of the International Computer Chess Association*, 18(1), March 1995.

[73] K. Langendoen, R.A.F. Bhoedjang, and H.E. Bal. Models for Asynchronous Message Handling. *IEEE Concurency*, 5(2):28–38, April–June 1997.

[74] K. Langendoen, J.W. Romein, R.A.F. Bhoedjang, and H.E. Bal. Integrating Polling, Interrupts, and Thread Management. In *Proceedings of Frontiers'96*, pages 13–22, Annapolis, MD, October 1996.

[75] M.E. Lesk and E. Schmidt. Lex - A Lexical Analyzer Generator. Technical Report 39, Bell Laboratories Computing Science, October 1975.

[76] T.A. Marsland and M. Campbell. Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys*, 14(4):533–551, December 1982.

[77] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161–175, 1987.

[78] B. Nichols, B. Buttlar, and J. Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Newton, MA, 1996.

[79] N.J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw–Hill, New York, NY, 1971.

[80] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, December 1995.

[81] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Harvester Wheatsheaf, New York, N.Y., second edition, 1993.

[82] B. Pell. Metagame: A New Challenge for Games and Learning. In H.J. van den Herik and L.V. Allis, editors, *Heuristic Programming in Artificial Intelligence*, volume 3, pages 237–251. Ellis Horwood Ltd, Chichester, West Sussex, 1992.

[83] B. Pell. A Strategic Metagame Player for General Chesslike Games. In *AAAI National Conference*, pages 1378–1385, July 1994.

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

[85] G.L. Peterson. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, 12:115–116, June 1981.

[86] A. Plaat. *Research, Re: Search & Re-Search*. PhD thesis, Erasmus Universiteit Rotterdam, the Netherlands, June 1996.

[87] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-First Fixed-Depth Minimax Algorithms. *Artificial Intelligence*, 87(1–2):255–293, November 1996.

[88] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Exploiting Graph Properties of Game Trees. In *AAAI National Conference*, pages 234–239, August 1996.

[89] C. Powley and R.E. Korf. Single-Agent Parallel Window Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(5):466–477, 1991.

[90] L. Prylli and B. Tourancheau. BIP: a New Protocol Designed for High Performance Networking on Myrinet. In *Workshop PC-NOW, IPPS/SPDP'98*, Orlando, FL, 1998.

[91] L. Prylli, B. Tourancheau, and R. Westrelin. Modeling of a High Speed Network to Maximize Throughput Performance: The Experience of BIP over Myrinet. In *Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, 1998.

[92] V. Rao, V. Kumar, and K. Ramesh. A Parallel Implementation of Iterative-Deepening-A*. In *AAAI National Conference*, pages 178–182, July 1987.

[93] A. Reinefeld. An Improvement of the Scout Tree-Search Algorithm. *Journal of the International Computer Chess Association*, 6(4):4–14, 1983.

[94] A. Reinefeld and T.A. Marsland. Enhanced Iterative-Deepening Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, July 1994.

[95] A. Reinefeld, J. Schaeffer, and T.A. Marsland. Information Acquisition in Minimal Window Search. In *International Joint Conference on Artificial Intelligence*, volume 2, pages 1040–1043, 1985.

[96] A. Reinefeld and V. Schnecke. AIDA* — Asynchronous Parallel IDA*. In *Canadian Conference on Artificial Intelligence*, pages 295–302, Banff, Canada, 1994.

[97] J.W. Romein, H.E. Bal, and D. Grune. Multigame — A Very High Level Language for Describing Board Games. In *First Annual ASCI Conference*, pages 278–287, Heijen, the Netherlands, May 1995.

[98] J.W. Romein, H.E. Bal, and D. Grune. An Application Domain Specific Language for Describing Board Games. In *Parallel and Distributed Processing Techniques and Applications*, volume I, pages 305–314, Las Vegas, NV, July 1997. CSREA.

[99] J.W. Romein, H.E. Bal, and D. Grune. The Multigame Reference Manual. Technical Report IR-475, Faculty of Sciences, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, the Netherlands, August 2000.

[100] J.W. Romein, A. Plaat, H.E. Bal, and J. Schaeffer. Transposition Driven Work Scheduling in Distributed Search. In *AAAI National Conference*, pages 725–731, Orlando, FL, July 1999.

[101] J. Schaeffer. Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[102] 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.

[103] J. Schaeffer. Search Ideas in Chinook. In H.J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 19–30. Universiteit Maastricht, the Netherlands, 2000.

[104] 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.

[105] J. Schaeffer, P. Lu, D. Szafron, and R. Lake. A Re-examination of Brute-force Search. In *Games: Planning and Learning: AAAI 1993 Fall Symposium, Report FS9302*, pages 51–58, Chapel Hill, NC, 1993.

[106] J. Schaeffer, P. Lu R. Lake, and M. Bryant. Chinook: The World Man-Machine Checkers Champion. *AI Magazine*, 17(1):21–29, 1996.

[107] M. Schijf, L.V. Allis, and J.W.H.M. Uiterwijk. Proof-Number Search and Transpositions. *Journal of the International Computer Chess Association*, 17(2):63–74, 1994.

[108] J.J. Scott. A Chess-Playing Program. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 255–265. Edinburgh University Press, 1969.

[109] D.B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[110] 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.

[111] A. Sparrowhawk. *LOGO: A Language for Learning*. Pan Books, London, 1984.

[112] U. Stern and D.L. Dill. Parallelizing the Murphi Verifier. In *Ninth International Conference on Computer Aided Verification*, pages 256–267, 1997.

[113] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk-4.1 (Beta 1) Reference Manual*, September 1996.

[114] 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.

[115] L. Taylor and R.E. Korf. Pruning Duplicate Nodes in Depth-First Search. In *AAAI National Conference*, pages 756–761, July 1993.

[116] K. Thompson. Computer Chess Strength. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 55–56. Pergamon Press, Oxford, 1982.

[117] J.D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *Principles of Distributed Computing*, pages 214–222, 1995.

[118] K. Verstoep, K.G. Langendoen, and H.E. Bal. Efficient Reliable Multicast on Myrinet. In *International Conference on Parallel Processing*, volume 3, pages 156–165, Bloomingdale, IL, August 1996.

[119] D.L. Weaver and T. Germond, editors. *The SPARC Architecture Manual*. Prentice Hall, Menlo Park, CA, 1994.

[120] J.-C. Weill. *Programme d'Échecs de Championnat : Architechture Logicielle, Synthèse de Fonctions d'Évaluations, Parallélisme de Recherche*. PhD thesis, Université Paris 8, January 1995. In French.

[121] J.-C. Weill. The ABDADA Distributed Minimax Search Algorithm. In *Proceedings of the 24th Annual ACM Computer Science Conference*, Philadelphia, PA, February 1996.

[122] 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.

Curriculum Vitae

Name:	Johannes Willem Romein
Born:	March 26th, 1970, Culemborg, The Netherlands
Nationality:	Dutch

Sep. 1982 – Jun. 1988	Atheneum B Hermann Wesselink College, Amstelveen
Sep. 1988 – Aug. 1994	Master's degree in Computer Science Vrije Universiteit, Amsterdam
Feb. 1994 – Aug. 1994	Student assistent (1 day per week) Vrije Universiteit, Amsterdam
Sep. 1994 – Aug. 1998	Ph.D. student Vrije Universiteit, Amsterdam
Sep. 1998 – Feb. 2000	Writing Ph.D. thesis (3.5 days per week) Vrije Universiteit, Amsterdam
Sep. 1998 – Feb. 2000	System administrator (1.5 days per week) Vrije Universiteit, Amsterdam
Mar. 2000 – present	Researcher Vrije Universiteit, Amsterdam

Present address:

Vrije Universiteit
Faculty of Sciences, Department of Mathematics and Computer Science
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands

email: john@cs.vu.nl

Publications in international, refereed conferences

[1] J.W. Romein and H.E. Bal. Parallel N-Body Simulation on a Large-Scale Homogeneous Distributed System. In *EuroPar '95 (LNCS 966)*, pages 473–484, Stockholm, Sweden, August 1995. Springer-Verlag.

[2] K. Langendoen, J.W. Romein, R.A.F. Bhoedjang, and H.E. Bal. Integrating Polling, Interrupts, and Thread Management. In *Proceedings of Frontiers'96*, pages 13–22, Annapolis, MD, October 1996.

[3] J.W. Romein, H.E. Bal, and D. Grune. An Application Domain Specific Language for Describing Board Games. In *Parallel and Distributed Processing Techniques and Applications*, volume I, pages 305–314, Las Vegas, NV, July 1997. CSREA.

[4] R.A.F. Bhoedjang, J.W. Romein, and H.E. Bal. Optimizing Distributed Data Structures Using Application-Specific Network Interface Software. In *International Conference on Parallel Processing*, pages 485–492, Minneapolis, MN, August 1998.

[5] J.W. Romein, A. Plaat, H.E. Bal, and J. Schaeffer. Transposition Driven Work Scheduling in Distributed Search. In *AAAI National Conference*, pages 725–731, Orlando, FL, July 1999.

Technical reports and national conferences

[1] J.W. Romein and H.E. Bal. Parallel N-Body Simulation on a Large-Scale Homogeneous Distributed System. Technical Report IR-364, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1994.

[2] J.W. Romein, H.E. Bal, and D. Grune. Multigame — A Very High Level Language for Describing Board Games. In *First Annual ASCI Conference*, pages 278–287, Heijen, the Netherlands, May 1995.

[3] J.W. Romein, H.E. Bal, and D. Grune. The Multigame Reference Manual. Technical Report IR-475, Faculty of Sciences, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, the Netherlands, August 2000.