Exploiting Fine-grain Parallelism in Concurrent Constraint Languages

Johan Montelius

A Dissertation to be submitted
for the Degree of Doctor of Philosophy
Computing Science Department
Uppsala University
Sweden

April 1997

Uppsala University
Computing Science Department
Box 311
SE-751 05 Uppsala, Sweden

Swedish Institute
of Computer Science
Box 1263
SE-164 29 Kista, Sweden

Uppsala Thesis in Computing Science 28
ISSN 0283-359X
ISBN 91-506-1215-8

SICS Dissertation Series 25
ISSN 1101-1335
ISRN SICS/D--25--SE

Doctoral thesis at Uppsala University 1997
from the Computing Science Department

Abstract

Montelius, J., 1997. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*, 220 pp. Uppsala Thesis in Computing Science 28, ISSN 0283-359X, ISBN 91-506-1215-8. SICS Dissertation Series 25, ISSN 1101-1335, ISRN SICS/D--25--SE

THIS DISSERTATION PRESENTS the *design*, *implementation*, and *evaluation* of a system that exploits fine-grain implicit parallelism in concurrent constraint programming language. The system is able to outperform a C implementation of an algorithm with complex dependencies without any user annotations.

The concurrent constraint programming language AKL is used as a source programming language. A program is divided during runtime into tasks that are distributed over available processors. The system is unique in that it handles both and-parallel execution of goals as well as or-parallel execution of encapsulated search.

A parallel binding scheme for a hierarchical constraint store is presented. The binding scheme allows encapsulated search to be performed in parallel. The design is justified with empirical data from the implementation. The scheme is the most efficient parallel scheme yet presented for deep concurrent constraint systems.

The system was implemented on a high-performance shared-memory multiprocessor. Extensive measurements were done on the system using both smaller benchmarks as well as real-life programs. The evaluation includes detailed instruction-level simulation, including cache-performance, to explain the behavior of the system.

Keywords Implicit parallelism, Concurrent Constraint Programming, Cache performance, Logic programming, Abstract machine, Parallel execution, Scheduling, Multiprocessor, Shared memory.

Johan Montelius, SICS, Box 1263, SE-164 29 Kista, Sweden

ISSN 0283-359X, 1101-1335
ISBN 91-506-1215-8
Printed in Sweden by Gotab, Stockholm 1997.

Till Morfar

Acknowledgments

This work is based on the design of the Agents Kernel Language and the implementation of the sequential AGENTS system. The design and implementation was the joint effort of a large group of people in the former Concurrent Constraint Programming group at SICS.

I would like to thank all of my colleagues for making this thesis possible and for providing a creative atmosphere. In the design and implementation of the parallel system I have worked with Dr. Khayri A. M. Ali, Dr. Galal Atlam and Haruyasu Ueda. Without their help there would not have been a complete parallel system. In the evaluation I have been aided by Peter Magnusson whose understanding of the target architecture helped me understand why Penny behaves as it does and how to find the bottlenecks. Without the use of the SimICS simulator many questions would still be open. During the whole process I have been coached by Prof. Seif Haridi. His enthusiasm is an inspiration for me.

I would also like to thank all my friends who have read drafts of this dissertation and helped me improve the presentation.

Table of Contents

List of Figures

List of Tables

Prologue

How long time have you been working with it? This is a question that I often get but find very hard to answer. Why? – well, it depends on how you want to look at it.

I wrote my first computer program on an ABC80 using BASIC more than fifteen years ago. I guess my interest in computers started then but it certainly had nothing to do with parallel computers. The environment at Uppsala University, where I started in 1984, inspired me to study logic programming. As a masters thesis I implemented an or-parallel Prolog system that I hope is forgotten by now ...oops! The Uppsala years can be defined as the start of my interest in the implementation of declarative programming languages.

In 1990 I was drafted to SICS and started to work with Sverker and Seif on the design of the AKL system. This was the start of my interest in implementations of concurrent languages. During the first couple of years there was no time to work on a parallel implementation, although this was something that we always had in mind. During 1993 I started on the design of the parallel system. From then on it was parallelism and Penny only.

Another question is – What is it about? This is easier to answer. It is all about time.

⋆ ⋆ ★ ⋆ ⋆

Chapter 1

Introduction

THIS DISSERTATION PRESENTS the *design*, *implementation*, and *evaluation* of a system that exploits fine-grain implicit parallelism in a concurrent constraint programming language. The system is able to outperform a C implementation of an algorithm with complex dependencies without any user annotations.

1.1 Exploiting parallelism

As computers with multiple processing units are becoming available, the need for tools that exploit their processing power will increase. The bottleneck today is not the hardware but the programming environments. It is hard to write programs that exploit the potential power of a multiprocessor since the parallelization must largely be done by hand.

Automatically exploiting parallelism in a program is difficult. An automatic system must divide the program into parts that can be executed in parallel. The parts should not only execute in parallel without any conflicts, but the result of the execution should be identical to that of a single-processor computer. Most automatic systems therefore rely on user annotations or restrict the parallelization to certain programming constructs.

In this dissertation I present a system that automatically exploits the power of a multiprocessor without the need for annotations from the programmer. The system exploits fine-grain parallelism available in a concurrent constraint programming language. A program is divided dynamically into tasks that can be distributed over the available processors. The system shows good parallel performance on a twenty processor shared-memory multiprocessors.

1.2 Contributions

This dissertation describes the design, implementation and evaluation of a parallel implementation of AKL. The three main contributions in this dissertation are:

- The design of a parallel execution model. The model defines how programs are executed. It allows multiple processors to collaborate in the execution while maintaining the efficiency of sequential execution. The model is justified and compared to alternative solutions.

- The design of a parallel binding scheme that allows for very efficient implementation of encapsulated search. This is the most complex component in a constraint language and allows encapsulated search to be performed in parallel. The scheme has advantages even in sequential systems since it keeps perfect track of stability. The design is justified with empirical data from the implementation.
- An extensive evaluation of the complete system. The evaluation includes smaller benchmarks that focus on particular aspects as well as larger real-life programs. The evaluation shows that the system works even for fine-grain computations. Cache performance and garbage collections are shown to be the limiting factors when running larger programs. The evaluation is partly done by using instruction-level simulation techniques, that model cache behavior, to reveal limitations and explain the behavior of the system.

The work presented in this dissertation is based on the development of the Agents Kernel Language (AKL). The language was developed by a large group of people led by Prof. Seif Haridi and Dr. Sverker Janson. The language was designed in a way that would allow exploitation of parallelism.

1.3 Publications

The material in the dissertation has partly been presented in the articles listed below.

Sverker Janson, Johan Montelius, and Seif Haridi. *Ports for objects in concurrent logic programs* In Agha, Wegner, and Yonezawa (eds.), Research Directions in Concurrent Object-Oriented Programming, The MIT Press, 1993

Johan Montelius and Khayri A. M. Ali. *An And/Or Parallel Implementation of AKL* In New Generation Computing, vol 14, no 1, Special issue on the Workshop on Parallel Logic Programming held in Portland, Oregon, November 1995.

Haruyasu Ueda and Johan Montelius. *Dynamic Scheduling in an Implicit Parallel System*, In the Proceedings of the ISCA 9th International Conference on Parallel and Distributed Computing Systems held in Dijon, France, September 1996.

Johan Montelius and Peter Magnusson. *Using SimICS to evaluate the Penny system* IJCLP'96, Parallel Implementation workshop held in Bonn, Germany, September 1996.

Johan Montelius and Seif Haridi. *An evaluation of Penny: a system for fine-grain implicit parallelism* to be presented at the Second International Symposium on Parallel Symbolic Computation, ACM SIGSAM/SIGNUM, July 1997.

The work has also been presented in the deliverables of the PEPMA and ACCLAIM Esprit projects.

1.4 Outline of the thesis

Chapter 2 gives the background to the work. The chapter explains why parallelism is difficult to exploit in traditional programming systems and how it can be exploited in concurrent constraint programming systems. The chapter also includes an overview of how modern shared-memory multiprocessors operate, and the special consideration that any implementation must take into account.

Chapter 3 is an introduction to AKL programming. Read this chapter and you will be able to write a parallel program.

Chapter 4 describes the computation model of AKL. The model is a formal description of the operational semantics of the language.

Chapter 5 presents the execution model. The model describes how AKL programs are executed in the Penny system. The chapter addresses things like eager/lazy execution, fairness, implicit/explicit parallelism etc. It also discusses alternative approaches and compares the system to other concurrent systems. Reading this chapter will give you an understanding of how an AKL program behaves when executed in the Penny system.

The implementation is described in Chapters 6, 7 and 8. These chapters contain many implementation details. The first chapter describes the abstract machine and its instructions set. The second describes the parallel binding scheme. The third is a presentation of implementation techniques used for representation of terms, memory allocation, copying procedures etc.

The evaluation is presented in Chapter 9. This is probably the first chapter that you will look at (my guess is that you have already tried to figure out if the system works by looking at the speedup diagrams). If you have read Chapter 3, and do not want to know how things work, this is the only chapter you need to read.

Chapter 10 describes related work in the area. This chapter also includes a performance comparison with similar systems.

The last chapter is a summary and a discussion of future work.

Chapter 2

Background

THERE IS ONLY ONE REASON to execute a program in parallel – time. Parallel execution can reduce the time it takes to perform computations, it will not add functionality to a system. From a user's point of view, parallelism will only mean less time. From a programmer's point of view it will normally mean more trouble.

2.1 Parallelism

A parallel computer can reduce the time it takes to execute programs either by decreasing the time it takes to execute a single program, or increasing the number of programs that can be executed during a certain time. The latter is called throughput and has been successfully solved, the former, which we normally refer to as parallel programming, is harder and has only recently been successfully applied [65].

2.1.1 Parallel programming

Parallel programming has been successful for large numerical applications on massive parallel machines. It is natural that this area has drawn the most attention since these applications have the most to gain from parallelism. The applications are computation intensive and can easily be distributed on the available processors. The drawback is that the types of applications are limited and that the hardware often is not suited for general-purpose programming.

General-purpose parallel programming has not made the same advances. The reason is that the potential speedup is one or two magnitudes lower. Given the limited speedup it is of course hard to justify dedicated hardware and more complex programming environments.

This is changing as small-scale shared-memory multiprocessors are becoming widely available. These multiprocessors are not developed with parallel programming as their main target application. Instead it is the sharing of resources and throughput that has been the driving force. The availability of machines will increase the demand on general purpose parallel programming technology. If the machines are there, we only have to tackle the complexity of parallel software development.

2.1.2 Implicit parallelism

Programming systems for parallel computers can be divided into two groups. There are systems that make the parallelism *explicitly* available to the programmer and there are systems that automatically exploit *implicit* parallelism that is available in a program. Since automatic parallelization is difficult, and since most interest has been directed to massive parallel machines, the explicit approach has been the most successful. This will change; as multiprocessors are becoming available we will need systems that handle the parallelization automatically.

The benefit of using implicit parallelism is that the programmer can concentrate on solving the problem, not on coding the solution for parallel execution. Coding for parallel execution is not trivial and requires, for optimal performance, knowledge of the hardware. A system that has been adapted for a particular hardware is always harder to understand and maintain. Moreover, coding for a particular hardware limits the portability of the code.

Although implicit parallelism has many advantages it will never be an alternative if it cannot compete in performance with the explicit systems. The question today is how to make the implicit systems competitive. This dissertation shows that it is feasible to build an efficient parallel system that exploits implicit parallelism. The system dynamically divides a program into tasks and distributes the tasks among available processors. The parallelization is done without any user annotations. This is hard to do in most languages but easy to do if the language is carefully designed.

2.2 Programming paradigm

Any computation can be described as operations performed on a *store* containing a set of boxes where each box contains a set of fields and each field holds either a *value* or an *address* of a box. The address of a box is like a key, without the address the content cannot be accessed. An operation can *create* a new box, manipulate the value of a field, or determine the next sequence of operations that should be executed.

2.2.1 Threads

A program is a description of a set of *threads*. A thread is a sequence of operations that should be executed. Exactly which operations that are executed is determined by the operations themselves. The only input to the operations is the content of the store.

If the language only allows one thread it is a sequential language. The execution of a thread in a sequential language is deterministic. Since the only input to the operations is the store, and the information in the store is determined by the sequence, the outcome of the computation is always the same.

If more than one thread is allowed the language is *concurrent*. The outcome of a concurrent system is not deterministic, it is dependent on the interaction between the threads. This makes concurrent programming more complex. The advantage is that a program can be divided into modules with separate responsibilities. This results in a more structured program. Concurrency also opens the possibility to execute the program in parallel. The threads can be distributed on the available processors and executed in parallel. The problem is of course to predict the outcome of the execution.

Programming languages can be described and categorized in many ways. In this introduction I make a distinction between *imperative* and *declarative* programming languages. The difference between imperative and declarative programming languages is which operations the languages provide to manipulate the store.

2.2.2 Imperative languages

In an imperative language the contents of the fields can be changed. One operation can write a value to a field but there is nothing that prevents another operation to write another value to the same field. Since the values of fields can change, the result of a program is dependent on the order in which the operations are executed.

In a sequential language this is not a problem since the thread determines the order of execution. In a concurrent language it causes problems and special constructs must be introduced to synchronize the execution.

The most primitive construct is called a *lock*. A lock can be taken, but only by one thread at a time. If one thread holds a lock other threads will have to wait for the lock to be released. The thread that holds the lock can perform operations that update the contents of boxes and can do so without being disturbed by other threads. The threads that are waiting for the lock know that it is unsafe to trust any information in the boxes unless the lock is taken.

To automatically parallelize an imperative program, the system must divide the program into threads that can be executed in parallel. This is hard since any boxes that are shared between threads must be protected by locks. Automatic parallelization has been successful for well-structured languages such as Fortran but the parallelism is normally limited to special loop constructs.

In some programming systems, such as C, there is no distinction between a value and an address. An address can be treated as a value and a value can be used as an address. There is not even a distinction between values and sequences of operations. The access to boxes is not supervised so a write operation to the fifth field of a four-field box overwrites the first field of the adjacent box.

The C programming paradigm is, although loved by programmers, the worst possible scenario for the designer of a parallel system. The reason is that it gives the programmer full freedom to change the contents of any field in the system. There is no general way to determine if operations on the store are independent. The problem is undecidable.

Exploiting implicit parallelism in a C program is very hard for the system developer but handling explicit parallelism can be even harder for the programmer. Even if a program is divided into threads that should be independent there is nothing that prevents one part of the program to interfere with other parts. A thread can change any field in the store without taking a lock. Parallel programming in such environments is often a trade-off between performance and complexity (debugging).

2.2.3 Declarative languages

The declarative programming languages solve the problem by imposing one simple restriction; it is not allowed to change the content of a field. Once the value of a field is determined it cannot be changed, but a field is allowed to be empty until the value is known.

In a concurrent system the empty fields can be used for synchronization. A thread that tries to read the value of an empty field is suspended until the value is determined. This means that the result of a program execution can be independent of the execution order of threads. Threads automatically suspend and wait until the information is available.

A problem does occur if two threads try to determine the value of the same field. In pure functional programming languages this is syntactically impossible. Only one thread can possibly determine the value of a field and the field is only temporarily empty until the value has been calculated.

Logic programming languages can handle boxes with empty fields. Empty fields can be *bound* to a value. Nothing prevents two threads from binding a field to the same value but if conflicting bindings are imposed the execution fails. This means that the outcome of binding operations is always the same regardless of their order; consistent bindings succeed and conflicting bindings fail.

The programming paradigm of the declarative languages makes them ideal for automatic parallelization. Programs can easily be divided into threads and since the execution order is irrelevant we can choose any order, or execute the threads in parallel.

2.2.4 Reality

In reality things are not that simple. Since the declarative languages have been developed on sequential machines and since the parallelism has not been important, the designers of the languages have relaxed the rules. It is for example possible, in most declarative languages, to change the contents of a field or determine if a field is empty. This is not only due to bad design but also reflects a need. It is necessary to have constructs in a language that can change the value of a field. It can also be necessary to recover from a conflicting bindings or determine if a field is empty.

The constructs that change the store can be introduced in a more or less elegant way. If the constructs are introduced without parallel execution in mind automatic parallelization becomes as difficult as it is for the imperative languages.

2.3 Concurrent Constraint Languages

Concurrent constraint programming languages were developed from research in logic programming [45, 55]. It unified two strands of research, constraint logic programming and concurrent logic programming.

2.3.1 Constraints

As explained earlier a declarative program can be viewed as operations that assign, or read, the values of fields. A more general description can be given if we view a program as operations that impose constraints on variables.

A constraint is a relation between variables or between variables and values. A variable can for example be constrained to be less than a value ($X < 5$) or equal to another variable ($X = Z$). Setting up constraints limits the possible values of variables without necessarily binding them to exact values.

A set of constraints make up a *constraint store*. The store is either *consistent* or *inconsistent* i.e. free from contradictions or containing a contradiction. The store is maintained by a constraint solver that detects any inconsistencies. When we add constraints to the store, the store can become inconsistent. This is detected by the constraint solver.

A constraint is *entailed* by a store if the constraint can be deduced from the information available in the store. The constraint $X < 5$ is for example entailed by a store that contains the constraint $X = 2$. The dependencies can be complex, the constraint $X < 10$ is for example entailed by the store

$$\{X = Y + 2, Y < 3\}$$

The constraint solver detects these dependencies and determines if a constraint is entailed.

When a constraint is added it might seem that the value of a variable is changed. This is not really true, the possible values have only decreased. This means that a constraint that is entailed by a constraint store is entailed even if new constraints are added. The constraint $X < 10$ is for example still entailed by the store even if we add the constraint $Y < 2$. Only if we remove constraints from the store (for example $Y < 3$) can a situation occur where an entailed constraint no longer is entailed. Removing a constraint is therefore not allowed.

Entailment can be used to synchronize threads. A thread can suspend its execution until a constraint is entailed by the constraint store. This is similar to suspending on an empty field but more flexible. The constraint can involve many variables with complex dependencies.

2.3.2 Problem solving

In constraint logic programming the constraint system is not used for communication between threads but to describe and solve complex problems. A problem is solved using constraints by building a store of constraints that describes the solution to the problem.

As we build the constraint store, we might run into a choice; either we should add one set of constraints or another set of constraints. If we have several alternatives the store can be duplicated. Each store is then given the alternative set of constraints. A store that becomes inconsistent is discarded as a solution. Stores that survive can be duplicated again as other choices are explored. When all constraints that describe the solution have been added the surviving stores contain the possible solutions to the problem.

Making the choices and duplicating the store is a called the *search process*. The process is very expensive but the cost of doing the search can be reduced if the important choices are tried first. The search process is therefore guided by the programmer; it is the programmer that decides in which order the choices should be tried.

2.3.3 Agents Kernel Language

In order to combine the problem solving procedure with communicating threads the search process has to be encapsulated. Each thread maintains its own set of local constraint stores that it is allowed to duplicate without interference from other threads. This means that each thread can do its own search. Communication between the threads is done in the store that is shared among the threads.

The local constraint stores can impose constraints on variables that belong to the shared constraint store but these constraints are not visible outside of the local store. This allows a thread to set up assumptions and to evaluate the consequences without changing the shared constraint store.

Once we have a notion of shared and local constraint store it is easy to let each local constraint store be a shared constraint store for local threads. These local threads can of course have their local constrain stores. The result is a hierarchy of constraint stores. A thread in a store sees the information in the store above it, but only adds information to the store that it shares with its siblings.

The Agents Kernel language (AKL) was the first language to include both thread communication and problem solving in a uniform framework. The problem solving is not performed in watertight compartments but are open to receive additional information from the constraint stores in which the search procedure is encapsulated. The design of AKL was not an easy task and the implementation was even harder. The problem is to maintain the hierarchy of constraint stores. To do a parallel implementation is even more difficult.

There are two sources for parallelism in an AKL execution. The first one is the threads that can be executed in parallel, this is referred to as and-parallelism. The second one is the alternatives in search procedures encapsulated in threads, this is referred to as or-parallelism.

2.4 The Hardware

A programming language should preferably be independent of the hardware architecture but this is not easy to achieve. It is easy to implement a language on just about any platform but it is not easy to make an efficient implementation on all platforms. Multiprocessors, especially, have different characteristics and these are reflected in the programming constructs of the language. A language that is targeted to one particular architecture might be impossible to implement efficiently on another architecture. It is therefore important to determine the target for the programming language.

2.4.1 Multiprocessors

The target architecture, in this thesis, is a shared-memory multiprocessor. Today this is the dominating parallel architecture. The reason for this is that it is a general purpose architecture that is not only designed for parallel execution but for sharing of resources. The architecture provides a flexible way of sharing resources such as I/O capacity and memory. Shared memory machines will be developed and bought even if no program will make use of more than one processor at a time.

Shared-memory machines are already widely available as file servers. The smaller systems have six to eight processors while the larger servers can have up to 128 processors. The shared-memory architecture is even challenging the super-computer segment with high-end computation servers. Clusters of such servers are likely to replace the traditional super computers [73].

The shared-memory architecture has also been introduced for personal computers. Two and four processor computers are available. These systems might become the standard configuration.

The competing parallel architectures are vector machines and message passing machines. These machines are specialized for providing high performance and are mainly used for numeric intensive programs or programs with regular patterns. Programs that do not fall into these categories do not benefit from the architecture.

Networks of workstations are also used as parallel machines but the applications that can benefit from these are also very special. The granularity of the subtasks must be very large to hide the comparably long latencies in the network.

2.4.2 Shared address space

The most important feature of a shared-memory architecture is not that the memory is physically shared among processors. The most important feature is that the architecture provides the programmer with a uniform address space.

The shared address space is a flexible programming environment. It is ideal for implementing parallel systems. All threads have equal access to all data structures so there is no need to separate the data into global data and local data, or keep track of which processor is responsible for which data. Programming a shared-memory machine is thus easy, but making an efficient system is difficult.

2.4.3 Caches

The rate of instruction execution in a modern processor is much higher than the rate of data that can be supplied from memory. Accessing the main memory is typically thirty to sixty times as expensive as doing a register operation. This difference is not likely to go away, memory will become faster but the speed of processors will also increase. The problem is referred to as the memory wall and will only increase in the future [74].

Since access to shared-memory is expensive, the processors are equipped with caches. A cache is an intermediate storage device between the processor and the memory. It is a fast memory where the processor can store a copy of the data structures it is working on. The processor can manipulate the data locally and only needs to write it back to the main memory when it is done with all operations. In a single-processor architecture the only problem is which data should reside in the cache. The cache cannot hold all the data that the processor needs so there must be a plan that determines which data that should be copied and which should be stored in memory.

In a multiprocessor architecture, things are not as simple. If two processors copy a data structure to their caches, change the content of the structure and then writes it back to the main memory, things can go wrong. For example, if two processors want to read the value of a data structure, add one to the value and store the new value in the data structure, the final value in the data structure can be either the original value plus one or plus two. There can be an inconsistency in the system. This is something that must be avoided.

To solve the consistency problem, a protocol governs the processors' accesses to memory. If a processor wants to change a data structure it is given a copy of the structure, but all other copies of the data structure must first be invalidated. A processor that finds its own copy invalidated must request a new copy before it can trust the information. On the other hand, a processor that knows that it has the only copy can update its contents without further delay.

Although the shared-memory multiprocessors provide the programmer with a uniform memory access architecture, there is a big difference in accessing a data structure that no other processor is interested in, compared to accessing a data structure that all processors use. A parallel system that does not take this into account will have severe problems obtaining speedup.

2.5 Penny

The Penny system is a parallel implementation of AKL on a shared-memory multiprocessor. The Penny system is the first system that can exploit both and-parallelism and encapsulated or-parallelism in a concurrent constraint language. The system exploits the parallelism by, during runtime, dividing the execution into tasks that are distributed over the available processors. The task are only created if a processor is idle. There is little overhead compared to a sequential system.

– Is it hard to write an AKL program that runs in parallel? Turn the page and find out.

Chapter 3

Programming using Penny

THE PENNY SYSTEM is a parallel implementation of the AKL. This chapter is an informal introduction to AKL programming. It describes how to write, compile and execute programms in the Penny system.

3.1 AKL programs

I here give a description of the syntax of the AKL. It is not formal nor complete. It is not intended to be a programmers manual, its only purpose is to give an introduction to the AKL. The syntax is used in example programs in the following chapters.

3.1.1 Terms

A term is a syntactical object that is used to represent rational trees i.e. possibly infinite tree structures. The domain of rational trees is the only data type used in the Penny system. Terms are built out of *variables*, *constants* and *functors*.

Variables are written with an initial uppercase letter

```
X, Y, Foo, ...
```

or with an initial underscore.

```
_x, _foo, ...
```

A single underscore is an anonymous variable. Each occurrence of an anonymous variable in an expression is treated as a unique variable.

A constant is either written with lower case letters

```
a, b, bar, ...
```

or as a number.

```
652, 8.7, -51, ...
```

Given a functor, written in lower case letters, we can construct *compound terms*.

```
f(a,b), g(X, h(1), c), ...
```

Compound terms can have arbitrary number of *arguments* where each argument is a term. Following an old tradition we use a special syntax for lists. A list is either empty

```
[]
```

or contains one or more elements.

```
[a], [f(a,b), 2], [1,2,3,4],...
```

We can also use a construct

```
[A|Rest]
```

to describe a list with `A` as its first element and `Rest` as the remaining, possibly empty, list of elements. The term `[a|[b,c]]` is thus equal to `[a,b|[c]]` ,`[a,b,c|[]]` and `[a,b,c]`.

3.1.2 Atoms

Atoms are the primitive building blocks of a program. There are two kinds of atoms: *constraint atoms* and *program atoms*. Constraint atoms

```
X = a,  f(1,2) = f(Y,Z), [1,2,3] = [H|T], ...
```

describe an equality between terms. Given the above constraints `X` is *bound* to `a`, `Y` to `1`, `Z` to `2`, `H` to `1` and, `T` to `[2,3]`.

Program atoms

```
zot, foo(X,Y),  bar(a,[1,2]), ...
```

are used much in the same way as procedure calls in any procedural language. Program atoms can also be referred to by their *name* that consist of the functor and the arity of the atom.

```
zot/0, foo/2,  bar/2, ...
```

A set of built-in procedures are available and are used in the program examples. Since this is not a users manual there is no need to list the built-ins. When used they either perform an obvious operation or are explained.

3.1.3 Procedures

Procedures are defined using clause syntax. A *clause* consist of a program atom, called the *head*, and two, possibly empty, sequences of atoms. The sequences of atoms are divided by a *guard operator*. The first sequence is called the *guard* and the second is called the *body* of the clause. The constraint atoms of the guard can be implicit in the head of the clause.

There are three guard operators: *conditional*, *commit* and *wait* (`->`,`|`, `?`). The meaning of the different guard operators is explained in Section 3.3.3.

A *definition* consists of a sequence of one or more clauses. The heads of the clauses all have the same name and the clauses all have the same guard operator. A definition defines a procedure with the same name as the name of the clauses.

A program is a set of definitions where every program atom occurring in the program is defined exactly once and there is one definition of the `main/0` procedure.

3.2 Execution

The execution of AKL programs is best explained by an example.

3.2.1 A producer

We first define a procedure `producer/2` i.e. a procedure with two arguments. The arguments are a number and a variable. The variable is bound to, as a result of calling the procedure, a list of numbers.

```
producer(0, List):-
    -> List = [].
producer(N, List):-
    -> List = [N|Rest],
       dec(N, N1),
       producer(N1, Rest).
```

The first clause states that if the first argument is equal to `0` then the `List` should be empty. The equality constraint is implicit in the head of the clause. If the first clause is not applicable (if `N` is not equal to `0`) the second clause is used. This clause states that, regardless of the number `N`, the list should start with `N` followed by the sequence that we get if we call `producer/2` with a value that is one less than `N` (the built-in `dec(N,N1)` binds `N1` to a number that is one less than `N`). Calling the procedure `producer(4,L)` produces the binding `L = [4,3,2,1]`.

3.2.2 A consumer

In the second definition we use a more complicated guard. The `consumer/3` procedure reads the elements of a list and adds them to its second argument. The total sum is returned in its third argument.

```
consumer([], S, Sum):-
    ->  Sum = S.
consumer([N|Rest], S, Sum):-
    ->  add(S, N, S1),
        consumer(Rest, S1, Sum).
```

The first clause states that if the list is empty the `Sum` should be equal to its second argument. If the first clause is not applicable the second clause is tried. This clause states that if the first argument is a list, containing at least one element `N` and possibly other elements called `Rest`, then the `Sum` can be calculated by adding `N` to its second argument, resulting in `S1`, and calling the procedure `consumer(Rest, S1, Sum)`. Calling the procedure `consumer([4,3,2,1], 0, Sum)` results in the binding `Sum = 10`. Note that the guard does not impose any bindings, it is a descriptions of constraints that must be fulfilled before the clause can be used. The `consumer/3` procedure can therefore not produce a list, only consume one.

3.2.3 The program

Given the definitions of `producer/2` and `consumer/3` we can create a program by adding the following definition.

```
main:- -> producer(10000,List), consumer(List, 0, Sum).
```

The program constructs a `List` of numbers, from 10000 to 1 and then compute the `Sum` of the elements.

To execute the program using the Penny system you need a computer, preferably a multiprocessor. If you need a copy of the system you should be able to find one at *http://www.sics.se/~jm/penny.html*

Assuming we have stored the program in a file called `test.akl` we can compile it with the Penny compiler using the `pac` command.

```
$ pac test
{compiling test.akl...}
{test.akl compiled}
```

Now we should have a file called `test.pam` that contains the instructions for the Penny abstract machine. It is readable, so do not hesitate to take a look at it.

Once we have compiled the program we can execute it using the Penny runtime system. The boot file is given as a `-b` argument to the system. We give the `-v` flag as an argument to make the system output some statistics.

```
$ penny -v -b test.pam
  :
{runtime 149 ms. }
```

The execution took 149 milliseconds.

3.2.4 Speedup

If you are lucky to run on a multiprocessor you can try giving a `-w` argument to the system. This argument specifies how many processors that should be used.

```
$ penny -v -w 2 -b test.pam
  :
{runtime 99 ms. }
```

Ahh! Your first parallel execution shows speedup. The total execution time is now only 99 milliseconds, not twice as fast but yet a significant improvement.

3.3 Carrying on

In the following sections we assume that the reader is familiar with logic programming languages.

An AKL execution is a reduction of atoms in an *environment* of constraints. A constraint atom is reduced simply by adding the constraint to the environment. A program atom is reduced using its definition. In the reduction it is replaced by the constraints of the guard and the atoms of the body of the clause. This is called *promotion*. The guard operator determines when a reduction is allowed.

3.3.1 The guard

The guard is executed in its own local environment. A guard is *solved* when all atoms in the guard have been reduced. What remains is the constraints in the local environment.

The constraints in the local environment are not visible outside the guard. Constraints on variables that are external to the guard can either be entailed by, consistent with or inconsistent with the external environment. If all the constraints on external variables are entailed by the external environment we say that the guard is *quiet*.

Note that for a guard to be quiet it is not sufficient that the constraints of the guard are consistent with the external environment, they must be entailed. If it cannot be determined if the constraints of the guard are entailed by, or inconsistent with, the external environment the guard is suspended.

A guard *fails* if the constraints of the guard are inconsistent with the external environment. A clause with a failed guard can not be used for the reduction step. When a guard is evaluated all of the constraint atoms in the guard are considered. As an example we can look at the program below.

```
main:- -> foo(A,A,B).

foo(X,Y,Z):- X = 1, Y = 2 -> Z = no.
foo(X,Y,Z):- -> Z = yes.
```

The evaluation of the first guard results in an inconsistent constraint store since it requires `A` to be equal both to `1` and `2`. The evaluation does not stop when it encounters the constraint `X = 1` that is consistent but not entailed by the external environment. It proceeds and detects the inconsistency introduced by the constraint `Y = 2`.

Note that the two constraints `X = 1` and `Y = 2` are consistent with the external environment if treated individually. To detect the inconsistency the system has to build a local environment where the implications of the constraints can be recorded. How these local environments should be handled in a parallel implementation is what this thesis is all about.

3.3.2 Deep guards

So far we have not used any complicated tests in the guards. More complicated test can be defined by using calls to user defined procedures inside the guards. This is called *deep guards*.

For example, if we want to check if an element occurs in a list before selecting the right clause we can define a procedure `check/2`

```
check(X, [X|_]):- -> true.
check(X, [_|Rest]):- -> check(X, Rest).
```

and then use the procedure in a guard.

```
foo(A,B):- check(this, A) -> bar(B).
foo(A,B):- -> zot(B).
```

This is more convenient than having to write a `check/3` procedure that returns a value that can be used to determine which clause to use.

```
check(X, [], Flag):- -> Flag = no.
check(X, [X|_], Flag):- -> Flag = yes.
check(X, [_|Rest], Flag):- -> check(X, Rest, Flag).

aux(B, yes):- -> bar(B).
aux(B, no):- -> zot(B).

foo(A,B):- -> check(this, A, Flag), aux(B, Flag).
```

Guards can contain arbitrary many user defined procedures, and the procedures themselves can be defined using deep guards.

3.3.3 Conditional, commit and wait

The conditional guard operator (`->`) requires that the guard is solved and quiet before a reduction is allowed. It also orders the clauses in a definition. The second clause is not allowed to be used unless the guard of the first clause has failed. The third clause must wait for the second clause etc. If we want an arbitrary clause to be selected once its guard is quiet then the commit guard operator (`|`) is used.

The traditional example when a commit guard is needed is when two streams of messages are merged. If a message has arrived on one stream it should be forwarded as soon as possible. The system should not be suspended waiting for a message to appear on the other stream.

```
merger([], Y, M):- | M = Y.
merger(X, [], M):- | M = X.
merger([H|T], Y, M):- | M = [H|M2], merger(T, Y, M2).
merger(X, [H|T], M):- | M = [H|M2], merger(X, T, M2).
```

The third guard operator, the wait guard (?), is different. A program atom defined with a wait guard can only be reduced if there is only one applicable clause and the clause is solved. There is no requirement that the guard should be quiet, the only requirement is that it is solved and that all other guards have failed. The usage of wait guards is explained in Section 3.4.

3.3.4 Ports

Communication between processes through variables is an elegant programming technique as long as there is one sender and one or more receivers. A many-to-one communication is harder to describe.

In AKL *ports* are used to implement many-to-one communication [35]. A port is a link to a stream of messages represented by a list. Messages are sent to the port but are read from the stream. The receiver of the messages need therefore only have access to the stream. The port synchronizes the access to the process. Each port is a unique term and can thus also be used as an identifier of a process.

Three program atoms are needed to handle ports: `open_port/2` creates a port and connects it to a stream, `send/3` sends a term to a port and `port/1` is used as a recognizer. The usage of ports is best explained by an example. The program atoms

```
.., open_port(P0,S), send(m1, P0, P1), send(m2, P1, P2), ..
```

creates a port `P0` and connects it to the stream `S`. The send operations suspend until their second argument are known and then send the message given by their first argument to the port. As soon as the message has been delivered the third argument is unified with the second argument.

This results in `S` being bound to `[m1,m2|_]` assuming that no other messages are sent to `P0` nor `P1`. Not that the order of `m1` and `m2` is controlled by the synchronization provided by the variable `P1`. The message `m2` cannot be sent unless the message `m1` has successfully been sent and received. If the synchronization was omitted the messages could appear in any order.

3.3.5 Abstractions

The Penny system also provides a syntax for higher order data structures called abstractions. An abstraction consists of a sequence of formal arguments and a program atom.

```
(X,Y)\foo(X,2,Y)
```

To use an abstraction we need a special program atom `apply/2` that takes an abstraction and a list of arguments as input. The program atom

```
apply((X,Y)\foo(X,2,Y), [1,3])
```

is then equal to the program atom

```
foo(1,2,3)
```

A process that maps an abstraction to each element of a list, producing a new list with all the results can be written:

```
map([], Abstr, Result):-
     ->  Result = [].
map([H|Rest], Abstr, Result):-
     ->  Result = [R|Map],
         apply(Abstr, [H,R]),
         map(Rest, Abstr, Map).
```

This procedure can then be used to produce a list of squares of all integers in a given list:

```
squares(List, Result):-
     ->  map(List, (X,R)/R is X * X, Result).
```

Using abstractions instead of Prolog like meta-call has many advantages. It for example syntactically decidable which definition that could possibly be used in an execution. This information could be used by a complier to remove or specialize code.

Abstractions are data structures that have a unique identity. Two abstractions although lexically identical are still treated as different. This is a pragmatic solution since it is in general undecidable if two abstractions denote the same AKL procedure.

3.4 Nondeterminate executions

Logic programming is most well known for its small puzzle solving programs. It is often very simple to write a small program that solves a complicated problem. AKL is not an exception but the technique is different from the one used in Prolog.

3.4.1 The Nobel prize

In 1979 Allan M. Cormak and Sir Godfrey received the Nobel prize in medicine for the invention of the Computed Axial Tomography (CAT) scanner. A CAT scanner can from a series of x-ray images, each taken from a different angle, make a cross-section image of a human body.

A recreational form of the CAT scanner was presented by A.K.Dewdney, the problem was simplified to the reconstruction of a pattern in a rectangular grid [19]. The squares in the grid could have only one of two values: filled or empty. The x-ray image was reduced to a image where each ray traverses a line in the grid. A line can be either horizontal, vertical or diagonal.

3.4.2 The scanner

The recreational form of the scanner is used to demonstrate different programming techniques in AKL. A small grid, only three by three squares is used in the examples, the puzzle of finding a pattern is not hard but serves as a good introduction to non-deterministic programming.

An image is represented as a list of integers. Each integer represents the number of filled squares in a line. A square is represented by one of the constants `on` or `off`. For simplicity only two images are used, one vertical and one horizontal. The grid is represented as a list of rows where each row is represented by a list of the squares in the row.

As an example we use a three by three grid. To write a simple AKL program that finds the pattern in the grid we first need a definition that describes the relation between the number of filled squares and the squares of the line. The relation is defined using the wait guard operator.

```
line(0, off, off, off):- ? true
line(1, on,  off, off):- ? true.
line(1, off, on,  off):- ? true.
line(1, off, off, on ):- ? true.
line(2, on,  on,  off):- ? true.
line(2, on,  off, on ):- ? true.
line(2, off, on,  on ):- ? true.
line(3, on,  on,  on ):- ? true.
```

3.4.3 Simple solution

The first problem is given by a horizontal image `[2,0,1]` and a vertical image `[1, 0, 2]`. The rays are ordered top to bottom and left to right. Using the `line/4` predicate we can write a procedure that finds the solution to our problem.

```
one([H1,H2,H3], [V1,V2,V3], Solution) :-
        line(H1,A1,A2,A3),
        line(H2,B1,B2,B3),
        line(H3,C1,C2,C3),
        line(V1,A1,B1,C1),
        line(V2,A2,B2,C2),
        line(V3,A3,B3,C3)
    ->  Solution = [[A1,A2,A3],[B1,B2,B3],[C1,C2,C3]].
```

When the procedure `one([2,0,1], [1,0,2], S)` is called, the system creates a local environment where the guard is evaluated. The first *goal* has more than one clause to choose from but for the second goal there is only one applicable clause. The fifth goal also has exactly one clause to chose from whereas the other goals have more than one.

If the second and fifth goals are resolved the bindings `B1 = off`, `B2 = off`, `B3 = off`, `A2 = off`,`C2 = off` are promoted to the local environment. The added information reduces the applicable number of clauses for the remaining goals. The first and last goals now have only one applicable clause each and are thus promoted. This resolves the final ambiguity and the guard is solved i.e. there are no remaining goals in the guard.

Since the guard does not bind any external variables it is also quiet and the solution `S = [[on,off,on][off,off,off],[off,off,on]]` can be promoted.

3.4.4 Choice splitting

What happens if none of the goals can be resolved? If we call the procedure `one([1,0,1],[1,0,1], S)` the system comes to a halt. The result after two reductions is equivalent to the procedure:

```
one([1,0,1], [1,0,1], Solution) :-
        line(1,A1,off,A3),
        line(1,C1,off,C3),
        line(1,A1,off,C1),
        line(1,A3,off,C3)
    ->  Solution = [[A1,off,A3],[off,off,off],[C1,off,C3]].
```

At this point none of the goals can be resolved, they all have two alternatives each. The system now does a choice split operation, it selects the first applicable goal and duplicates the guard. The first copy takes care of the first alternative and the second copy takes care of the remaining alternatives. Since both goals now have only one applicable clause they are resolved. The result is equivalent to the procedure:

```
one([1,0,1], [1,0,1], Solution) :-
        line(1,C1,off,C3),
        line(1,on,off,C1),
        line(1,off,off,C3)
    ->  Solution = [[on,off,off],[off,off,off],[C1,off,C3]].
one([1,0,1], [1,0,1], Solution) :-
        line(1,C1,off,C3),
        line(1,off,off,C1),
        line(1,on,off,C3)
    ->  Solution = [[off,off,on],[off,off,off],[C1,off,C3]].
```

Given the extra information the other goals now only have one alternative each and are consequently resolved. Since the first guard is solved and quiet it is promoted resulting in the binding `S = [[on,off,off],[off,off,off],[off,off,on]]`. The second solution is lost since we have encapsulated the puzzle in a conditional guard. In order to find all the solutions we use an aggregate construct.

3.4.5 Aggregates

An aggregate is a construct that collects all solutions to a goal. It can for example produce a list of all solutions or simply count them. In the Penny system one can create arbitrary aggregate procedures but there are some that are provided by the system.

The aggregate atom `bagof/2` takes two arguments, an abstraction with one argument and a term that is bound to list of all solutions.

A procedure that can generate all solutions is easily defined, we simply omit encapsulating the computation in a conditional guard.

```
solve([H1,H2,H3], [V1,V2,V3], Solution) :-
  ->  line(H1,A1,A2,A3),
      line(H2,B1,B2,B3),
      line(H3,C1,C2,C3),
      line(V1,A1,B1,C1),
      line(V2,A2,B2,C2),
      line(V3,A3,B3,C3)
      Solution = [[A1,A2,A3],[B1,B2,B3],[C1,C2,C3]].
```

The `one/3` procedure can be rewritten using the definition of `solve/3`.

```
one(Horizontal, Vertical, Solution):-
        solve(Horizontal, Vertical, S)
    ->  Solution = S.
```

If we are looking for all solutions we can define the procedure `all/3`

```
all(Horizontal, Vertical, Solution):-
    ->  bagof((X)/solve(Horizontal, Vertical, X), L).
```

A call to the procedure `all([1,0,1],[1,0,1],L)` results in the a binding of `L` to a list of all solutions of the scanner puzzle. If `numberof/2` is used the result is the binding `L = 2`.

3.5 The Penny system

The Penny system is based on the sequential AKL implementation called AGENTS. The Penny system does however only implement a subset of the syntax and functionality provided by the AGENTS system. The syntactical restrictions do not limit the usefulness of the system as research platform.

The AGENTS system includes a finite domain constraint solver but this has not been ported to the Penny system [12]. The system also included a feature called auto-closing. When all references of a port were lost, the stream of messages was closed i.e. bound to `[]`. This is an attractive feature but is not easily implemented and has not been ported to the Penny system.

The Penny system consists of a compiler and a runtime system. There is no interactive top-loop where programs can be incrementally compiled and loaded nor is there any debugger for the system. The parallelism in a Penny execution does not extend the functionality of the system. The only observable difference should be reduced execution time.

Chapter 4

The Computation Model

THIS CHAPTER DESCRIBES the computation model of the Agents Kernel Language. The computation model gives us an operational semantic that we need to discuss aspects of the language.

4.1 The computation model

The computation model gives AKL an operational semantics. This semantics is the one which defines the language. It is used as a reference when the execution model is described. The execution model and implementation should of course be sound with respect to the computation model but they are not complete. Since different execution models can have different behavior it is important to have a description of the language that is not influenced by a particular implementation. The operational semantics is also important when alternative language constructs are discussed.

The Penny computation model is based on the description of AKL in [32]. The semantics of AKL has also been described in [33], and [23]. The Penny computation model is simplified compared to the original AKL computation model. The original model is more general and is better suited when alternative language constructs is discussed. We have narrowed the model to only include the concepts that we need to describe the semantics of the Penny system. The description is also closer to the actual implementation. This makes the transition easier to the execution model and abstract machine.

The Penny system includes features not addressed by the original computation model, most notable are the abstractions and ports. We have tried to include these in the computation model although some aspects of higher order constructs are not easily described. The description of these features is close to how they are actually implemented.

The Penny system is restricted to only handle a syntactical subset of the AKL language. AKL allows nested definitions and explicit hiding of variables. These constructs have been removed from the language to simplify the execution model.

The restrictions do not decrease the expressiveness of the language, any AKL program can be rewritten in the supported subset.

I first describe an abstract syntax of the programming language and then how to represent the computation state and the rewrite rules. The final sections in this chapter are discussions of the properties of the language and possible alternative approaches.

4.2 Syntax

We need four sets of name spaces: *variables names*, *symbol names*, *program names*, and *abstraction names*. The symbol names are divided into two parts, a set of *functor names* and a set of *port names*. The port names never occur in a program but are used to describe how ports are handled in a computation.

4.2.1 Terms

From the name spaces we build the set of terms that are used to represent the domain of rational trees. The set of terms consist of *variables*, *atomic terms*, *compound terms* and *abstractions*.

$$
\begin{array}{lcl}
\langle \mathit{term} \rangle & ::= & \langle \mathit{variable\ name} \rangle \\
& & \mid \langle \mathit{functor\ name} \rangle \\
& & \mid \langle \mathit{functor\ name} \rangle(\langle \mathit{sequence\ of\ terms} \rangle) \\
& & \mid \langle \mathit{abstraction\ name} \rangle(\langle \mathit{sequence\ of\ variables} \rangle, \langle \mathit{program\ atom} \rangle)
\end{array}
$$

The abstraction names are only used in order to be able to define an equality relation over abstractions. Two abstractions are equal if they have the same abstraction name. This definition in conjunction with restrictions on the programs saves us from all the problems of handling equality between higher-order objects.

4.2.2 Atoms

There are two kinds of atoms, *program atoms* and *constraint atoms*.

$$
\begin{array}{lcl}
\langle \mathit{program\ atom} \rangle & ::= & \langle \mathit{program\ name} \rangle(\langle \mathit{sequence\ of\ variables} \rangle) \\
\langle \mathit{constraint\ atom} \rangle & ::= & \langle \mathit{term} \rangle = \langle \mathit{term} \rangle
\end{array}
$$

The variables of a program atom are distinct. The original AKL computation model is defined over arbitrary constraints in a first order constraint language but the Penny system is restricted to handle equality over the domain of rational trees.

4.2.3 Expressions

An expression is either a program atom, a constraint atom, a composition of expressions or one of the special expressions which handles ports, abstractions and aggregates. There are no nested choice expression nor explicit hiding of variables as in the original AKL model.

Expressions

$$
\begin{array}{lcl}
\langle expression \rangle & ::= & \langle program\ atom \rangle \\
 & & \mid \langle constraint\ atom \rangle \\
 & & \mid \langle expression \rangle, \langle expression \rangle \\
 & & \mid open\_port(\langle variable\ name \rangle, \langle variable\ name \rangle) \\
 & & \mid send(\langle variable\ name \rangle, \langle variable\ name \rangle) \\
 & & \mid apply(\langle variable\ name \rangle, \langle sequence\ of\ variables \rangle) \\
 & & \mid aggregate(\langle variable\ name \rangle, \langle variable\ name \rangle)
\end{array}
$$

4.2.4 Definitions

A definition consists of a program atom, called the *head*, and a *choice statement*. A choice statement consists of a sequence of *clauses* all with the same *guard operator*.

$$
\begin{array}{lcl}
\langle definition \rangle & ::= & \langle program\ atom \rangle := \langle sequence\ of\ clauses \rangle \\
\langle clause \rangle & ::= & \langle sequence\ of\ variables \rangle : \langle expression \rangle \langle pgo \rangle \langle expression \rangle
\end{array}
$$

Each clause consists of a set of variables and two expressions separated by a program guard operator. The first expression is the *guard* of the clause and the second is the *body* of the clause.

There are three types of program guard operators

$$\langle pgo \rangle \ ::= \ \rightarrow \mid \ '|' \mid ?$$

conditional, *commit* and *wait*.

4.2.5 Programs

When reasoning about programs we use the meta-variables listed in Table 4.1.

For a definition to be well formed, each variable occurrence should be *bound*. A variable occurrence is bound in a definition if

- the variable also occurs in the head of the definition,

symbol	concept
u, v, x, y, z	variable names
$\overline{u}, \overline{v}, \overline{x}, \overline{y}, \overline{z}$	sequences of variables
f	symbol name
p	program name
q	abstraction name
t	terms
a	program atom
c	constraint atom
E	expression
C	clause
$\%$	guard operator
D	definition

Table 4.1: Meta-variables for expressions

- it is an occurrence in E_1 or E_2 in the clause $\overline{x} : E_1 \% E_2$ and the variable also occurs in the sequence $\overline{x}$, or
- it is an occurrence in a in the abstraction $q(\overline{x}, a)$ and the variable also occurs in the sequence $\overline{x}$.

I assume that the notion of substitutions in expressions, where bound variables are renamed to make a substitution admissible, is known.

A *program* is a finite set of well-formed definitions of different program names, where every program name that occurs in the program has a corresponding definition and each abstraction has a unique abstraction name.

The requirement that all program atoms must have a definition simplifies the computation model since an exceptional case can be ignored. The requirement that all abstractions should have unique names makes it possible to define an equality theory.

4.3 The configuration

A configuration is a representation of a computation state. It is a tree-like structure constructed of *goals*.

4.3.1 Goals

There are three types of goals *local goals*, *global goals* and *guarded goals*, and two types of combinators *and-boxes* and *choice-boxes* in addition to program expressions.

symbol	concept
B, E	program expressions
σ, θ, ϕ	constraints
X, Y, Z, U, V	sets of variables
G	goal
S	sequence of local goals
R	sequence of guarded goals
$\%$	guard operator

Table 4.2: Meta-variables for configurations

We also use sequences of goals and guarded goals. A sequence is, a possibly empty, ordered comma-separated list.

Goals

$$
\begin{array}{lcl}
\langle goal \rangle & ::= & \langle global\ goal \rangle \mid \langle local\ goal \rangle \\
\langle global\ goal \rangle & ::= & \langle and\text{-}box \rangle \mid \mathbf{fail} \\
\langle and\text{-}box \rangle & ::= & \mathbf{and}(\langle sequence\ of\ local\ goals \rangle)^{\langle constraint \rangle}_{\langle set\ of\ variables \rangle} \\
\langle local\ goal \rangle & ::= & \langle choice\text{-}box \rangle \mid \langle expression \rangle \\
\langle guarded\ goal \rangle & ::= & \langle global\ goal \rangle \langle cgo \rangle \langle expression \rangle \\
\langle choice\text{-}box \rangle & ::= & \mathbf{choice}(\langle sequence\ of\ guarded\ goal \rangle)
\end{array}
$$

The description of goals is different from the description given by the original computation model. There are no *or-boxes* nor sequences of and-boxes. The description presented here is closer to the actual implementation. Aggregates are also handled differently compared to the original computation model. No explicit aggregate-box are used, instead a port like object are used in conjunction with two extra guard operators to mimic the execution of aggregates.

$$
\begin{array}{lcl}
\langle cgo \rangle & ::= & \langle pgo \rangle \\
 & & \mid collect(\langle variable\ name \rangle, \langle variable\ name \rangle, \langle variable\ name \rangle) \\
 & & \mid unit(\langle variable\ name \rangle, \langle variable\ name \rangle)
\end{array}
$$

Table 4.2 describes the meta-variables that are used in the description of the computation model. An empty and-box $\mathbf{and}()^{\theta}_{X}$ is called *solved* and is sometimes written θ_X. We also refer to solved guards.

The root of a configuration is a *global goal*, that is either an *and-box* or the symbol **fail**. An initial configuration holds one and-box called the *main and-box*. The main and-box initially contains a single program atom called the *main program-atom*.

4.3.2 Constraints

Constraints are formulas in a first order constraint language that, in Penny, is restricted to equality over terms, closed under conjunction and existential quantification.

In the following, $\exists\sigma$ stands for the existential closure of σ, and $\exists X$, where X is the set $\{x_1, \ldots, x_n\}$, for (any permutation of) the quantifier sequence $\exists x_1 \cdots \exists x_n$. We allow X to be empty, in which case $\exists X$ is mere decoration.

We assume given a complete and consistent constraint theory **TC** defining the following logical properties of constraints:

- σ is *satisfiable* iff $\mathbf{TC} \models \exists\sigma$,
- σ is *quiet* with respect to θ and V iff $\mathbf{TC} \models \theta \supset \exists V\sigma$
- σ and θ are *incompatible* iff $\mathbf{TC} \models \neg(\theta \wedge \sigma)$

The symbol **true** is used to denote a variable-free atomic formula for which it holds that $\mathbf{TC} \models \mathbf{true}$. It is also assumed that there is a variable-free atomic formula **false** for which $\mathbf{TC} \models \neg\mathbf{false}$. No further assumptions concerning the properties of the constraint theory are made unless explicitly stated.

4.3.3 Contexts

First we need the concept of *context*, that will allow us to define where goal rewrites are possible. A *context*, where λ denotes a *hole*, and χ denotes a context, is defined as follows:

- λ is a context.
- if χ is a context then $\mathbf{and}(S_1, \chi, S_2)_V^\theta$, $\mathbf{choice}(R_1, \chi, R_2)$ and $\chi\%E$ are contexts

A context is thus a configuration were exactly one local, guarded, or global goal is replaced by a hole.

Many of the rewrite rules place restrictions on the *environment* of the transformed component. Environments are defined in terms of contexts as follows.

- $env(\lambda) = \mathbf{true}$
- $env(\mathbf{and}(S_1, \chi, S_2)_V^\theta) = env(\chi) \wedge \theta$
- $env(\mathbf{choice}(R_1, \chi, R_2)) = env(\chi)$
- $env(\chi\%E) = env(\chi)$

An environment of a context is thus the conjunction of constraints in all and-boxes above the hole of the context.

The construct $\chi[G]$ denotes the goal obtained by substituting a goal G for λ in χ. The context χ can be considered as the context of this goal. The context $\chi[\chi']$ denotes the context obtained by substituting a context χ' for λ in χ.

4.4 The rewrite rules

An AKL *goal transition system*, with respect to a given program, is a structure $\langle \Delta, \Rightarrow \rangle$, where Δ is the set of expressions generated by $\langle goal \rangle$, and $\Rightarrow \subseteq \langle \Delta, C, \Delta \rangle$ is a transition relation, where C is the set of contexts.

Read

$$G \overset{\chi'}{\Rightarrow} G'$$

as saying that there is a transition from G to G' with context χ.

The *sub-goal* rule

$$\frac{G \overset{\chi'[\chi]}{\Rightarrow} G'}{\chi[G] \overset{\chi'}{\Rightarrow} \chi[G']}$$

states that we can rewrite a goal $\chi[G]$ in a context χ' by rewriting the sub-goal G in the context $\chi'[\chi]$.

In the rewrite rules we treat the constraints of an and-box as a conjunction where the $\wedge$ operator is associative and commutative but not idempotent. This allows us to select and remove individual constraints in a conjunction.

The rewrite rules consist of a set of *determinate* rules and one *non-determinate* rewrite rule (*choice splitting*).

4.4.1 Rules for program expression

Since we have limited the language to only allow definitions consisting of a head and a choice expression, we collapse the AKL program atom rule and choice rule in one rule.

The *program atom* rule is thus written

$$a \overset{\chi'}{\Rightarrow} \mathbf{choice}(\mathbf{and}(E_1)^{\text{true}}_{U_1}\%B_1, \ldots, \mathbf{and}(E_n)^{\text{true}}_{U_n}\%B_n)$$

and unfolds a program atom a using the definition $a ::= (U_1 : E_1\%B_1, \ldots, U_n : E_n\%B_n)$. The variables U_i are chosen to be disjoint from each other, from any set of variables in χ and from any set of variables bound by abstractions in E_i or B_i.

The *constraint atom* rule

$$\mathbf{and}(S_1, c, S_2)^{\theta}_{V} \stackrel{\chi}{\Rightarrow} \mathbf{and}(S_1, S_2)^{\theta\wedge c}_{V}$$

moves a constraint atom c to the constraint store of the and-box, thereby making it part of the environment of the goals in the and-box.

The *composition* rule

$$\mathbf{and}(S_1, (E_1, E_2), S_2)^{\theta}_{V} \stackrel{\chi}{\Rightarrow} \mathbf{and}(S_1, E_1, E_2, S_2)^{\theta}_{V}$$

creates two local goals from a composition.

The *open_port* rule

$$open\_port(x, y) \stackrel{\chi}{\Rightarrow} x = f(y)$$

creates a new constraint atom. The symbol name f is a port name that does not occur in χ. The constraint binds x to a unique port connected to the stream of messages y.

The *send* rule

$$\mathbf{and}(S_1, send(z, x), S_2)^{\theta\wedge x'=f(y)}_{V} \stackrel{\chi}{\Rightarrow} \mathbf{and}(S_1, S_2)^{\theta\wedge x'=f(y')\wedge y=[z|y']}_{V}$$

adds a message to the stream of messages associated with the port. It requires that $env(\chi) \cup \theta \models x = x'$ and that f is a port name. Note that the send rule replaces the constraint $x' = f(y)$ by the constraint $x = f(y') \wedge y = [z|y']$. The constraint store is thus not monotonic. It is possible to describe the send rule and keep the monotonic property of the constraint store but this requires that we use a more powerful constraint theory than equality over rational trees [32].

The *apply* rule

$$apply(x, \overline{y}) \stackrel{\chi}{\Rightarrow} p(\overline{v'})$$

requires that $env(\chi) \models x = q(\overline{u}, p(\overline{v}))$. The variables $\overline{u}$ in $p(\overline{v})$ are substituted by $\overline{y}$ giving the new expression $p(\overline{v'})$. The arity of $\overline{y}$ and $\overline{u}$ must of course be the same.

4.4.2 Pruning

Depending on the guard operator, the *pruning rule*

$$\mathbf{choice}(R_1, \theta_V\%B, R_2) \stackrel{\chi}{\Rightarrow} \mathbf{choice}(\theta_V\%B)$$

removes a guarded goal from the configuration. The constraint θ must be quiet with respect to the $env(\chi)$ and the set of variables V. The commit guard operator allows pruning in both directions while the conditional guard operator only allows pruning of the siblings to the right i.e. R_1 must be empty.

4.4.3 Promotion

If a choice-box has a single guarded goal with a solved guard as its only child, the body of the guarded goal can be promoted. The *promotion rule*

$$\mathbf{and}(S_1, \mathbf{choice}(\phi_U \% B), S_2)_V^\theta \overset{\chi}{\Rightarrow} \mathbf{and}(S_1, B, S_2)_{V \cup U}^{\theta \wedge \phi}$$

replaces a choice-box with the body of the promoted body. The conditional and commit guard requires that ϕ is quiet with respect to $env(\chi) \wedge \theta$ and U.

4.4.4 Failure

The *environment failure* rule

$$\mathbf{and}(S)_V^\theta \overset{\chi}{\Rightarrow} \mathbf{fail}$$

fails an and-box if θ and $env(\chi)$ are incompatible. The *guard failure* rule

$$\mathbf{choice}(R_1, \mathbf{fail}\ \%E, R_2) \overset{\chi}{\Rightarrow} \mathbf{choice}(R_1, R_2)$$

propagates the failure of an and-box to the guarded goal. The guarded goal is removed from the sequence of goals in a choice-box.

The *choice failure* rule

$$\mathbf{choice}() \overset{\chi}{\Rightarrow} \mathbf{false}$$

rewrites a choice-box with an empty sequence of guarded goals to a **false** constraint.

4.4.5 Non-determinism

The *choice splitting* rule

$$\begin{gathered}\mathbf{and}(S_1, \mathbf{choice}(\theta_V\ ?B, R), S_2)_U^\phi \% E \overset{\chi}{\Rightarrow} \\ \mathbf{and}(S_1', \mathbf{choice}(\theta_{V'}'\ ?B'), S_2')_{U'}^{\phi'} \% E', \mathbf{and}(S_1, \mathbf{choice}(R), S_2)_U^\phi \% E\end{gathered}$$

that is the only non-determinate rewrite rule, requires that the and-box $\mathbf{and}(S_1, \mathbf{choice}(\theta_V\ ?B, R), S_2)_U^\phi$ is *stable* or is a subgoal of a stable and-box. We define stability in Section 4.6.

The goals S_1', S_2', expressions B', E', and constraint ϕ' are obtained from the goals S_1, S_2, expressions B, E, and constraint ϕ by substituting the variables U for the variables U'. The variables U' are chosen to be distinct from any variables in U, S_1, S_2, R and χ. This is done in order to allow several guarded goals to be promoted without conflicts.

4.4.6 Aggregates

The *aggregate rules* are derived from the *aggregate scheme*

$$\mathbf{and}(S_1, aggregate(x,y), S_2)_U^\theta \overset{\chi}{\Rightarrow}$$
$$\mathbf{and}(S_1, \mathbf{choice}(G_1, G_2), S_2)_{U\cup z}^{\theta\wedge z=f(y)}$$

where the aggregate expression is removed and replaced by a new choice-box containing the guarded goals G_1 and G_2. A constraint $z = f(y)$ is also added to the and-box. The symbol name f is a port name that does not occur in χ nor in θ. The variable z is selected to be distinct from all variable in χ, θ, S_1 and S_2.

The guarded goal G_1

$$\mathbf{and}(apply(x,v))_{V_1}^{\mathrm{true}} collect(z,u,u')E_1$$

collects the solutions of the abstraction x. The abstraction is applied to a variable v and the result is collected using the the expression E_1 and the variables $\{u, u'\}$. The set of variables V_1 is constructed by the union of the variables in E_1 and the set $\{v, u, u'\}$.

The guarded goal G_2

$$\mathbf{true}_{V_2} unit(z, u'')E_2$$

will close the collection using the expression E_2 and the variable u''. The set V_2 is constructed from the variables in E_2 and the set $\{u''\}$.

The variables $\{v, u, u', u''\}$ are chosen to be distinct from all variables in χ, θ, S_1 and S_2. The expressions E_1 and E_2 are called the *collect expression* and the *unit expression* and are responsible for collecting the solutions once the branches are promoted.

The *collect rule*

$$\mathbf{and}(S_1, \mathbf{choice}(\phi_V collect(z,u,u')E, R), S_2)_U^{\theta\wedge z=f(y)} \overset{\chi}{\Rightarrow}$$
$$\mathbf{and}(S_1, E, \mathbf{choice}(R), S_2)_{U\cup V}^{\theta\wedge\phi\wedge z=f(u')\wedge u=y}$$

promotes one solution to the parent and-box if ϕ is quiet with respect to $env(\chi) \wedge \theta$ and V.

The *unit rule*

$$\mathbf{and}(S_1, \mathbf{choice}(\phi_V unit(z,u)E), S_2)_U^{\theta\wedge z=f(y)} \overset{\chi}{\Rightarrow}$$
$$\mathbf{and}(S_1, E, S_2)_{U\cup V}^{\theta\wedge u=y}$$

closes the aggregate if ϕ is quiet with respect to $env(\chi) \wedge \theta$ and V.

These two rules allow us to handle different types of aggregates. A construct similar to the traditional Prolog `bagof/2` can be described as an aggregate rule with the collect expression E_1 being $u = [v|u']$ and the unit expression E_2 being $u = []$. A `numberof/2` construct that only counts the number of solutions is easily defined as an aggregate with collect expression $add(u', 1, u)$ and a unit expression $u = 0$. Defining a `d-bagof/2` construct that collects the solutions in a difference-list is left as an exercise to the reader.

We can also have unordered aggregate rules that collect the solutions in any order. The only difference is that the collect rule does not require that the collected guarded goal is the leftmost.

4.5 A computation

An AKL computation is a derivation from an initial configuration using the described rewrite rules. A computation can be finite or infinite.

The initial configuration is an and-box

$$\mathbf{and}(p)^{true}_{\{\}}$$

with a single, variable free, program atom, a **true** constraint and an empty set of variables. The and-box is called the *main and-box* and the program atom is called the *main program atom*.

If a computation terminates the final configuration is either **fail**, θ_V or a non-solved and-box. The computation has then failed, succeeded or deadlocked.

Note that the main and-node can never be subjected to choice splitting. The choice splitting rule is only applicable to guarded goals not to an and-box.

4.6 Stability, choice splitting and candidates

The choice splitting rule rewrites a choice-box

$$\mathbf{choice}(R_1, \mathbf{and}(S_1, \mathbf{choice}(\theta_V\ ?B, R), S_2)^{\phi}_{U} \% E, R_2)$$

in context χ. It requires that the and-box

$$\mathbf{and}(S_1, \mathbf{choice}(\theta_V\ ?B, R), S_2)^{\phi}_{U}$$

is stable or is a subgoal of a stable and-box.

An and-box G is stable in the configuration $\chi[G]$ iff no determinate rewrite rule is applicable in G even if constraints, compatible with the environment of χ, are added to χ.

The first part is easy to understand but what does the "even if" part state? Why would a determinate rewrite rule suddenly become applicable when a constraint is added to χ and why are only compatible constraints considered?

Some determinate rules require that the constraint of an and-box is either quiet or incompatible with the environment of the context. If an and-box contains a constraint that is not quiet nor incompatible with respect to the environment a new constraint, that is added to the context, might push it in either direction and thus trigger a determinate rule.

We are only interested in constraints that are added to χ. Constraint can not be added to the and-box itself since only determinate rewrite operations in the and-box can add constraints to the and-box and those could by the first requirement not be performed. Furthermore we are only interested in constraints that are compatible with the environment of χ since if an unsatisfiable constraint is added to an and-box in χ the and-box can be rewritten to **fail**. The question of whether to do a choice splitting operation then becomes irrelevant.

Why does the split rule require that the and-box should be stable? There are actually two reasons for this. The first one is that we do not want to expand the computation unnecessarily, the second has to do with confluence.

4.6.1 Avoiding expansion

The choice split rule will duplicate some subgoals (S_1 and S_2 in the description above). This must be avoided until all other possibilities have been tried. Further computations might lead to a situation were the choice splitting rule is avoided. This is clearly desirable. If we postpone choice splitting until an and-box is stable, we know that no operation in the configuration can save us. The choice splitting rule will be the last chance for the stable and-box to contribute to the execution.

The choice splitting rule does not always make a contribution to the execution, the resulting and-boxes might be pruned or the environment might become inconsistent. On the other hand, choice splitting is always harmless. The operation will not limit the result of a computation, only make computations possible.

4.6.2 Confluence

The Penny computation model is not confluent. It is easy to see that the commit guard operator introduces indeterminism i.e. that the final configuration is depending on the way the pruning operation is performed. This also holds for unordered aggregates. The send rule also introduces indeterminism since it is sensible to the order in which the rules are applied.

These are however not the only ways in which indeterminism could be introduced. The guarded goal $\theta_V ?\, B$ in the description above is called the candidate of the stable and-box. Indeterminism can occur if the selection of the candidate is done in an unpredictable way. The reason for this is that the conditional guard operator is depending on the order of the guarded goals. Given an and-box

$$\mathbf{and}(\mathbf{choice}(G_1, R_1), \mathbf{choice}(G_2, R_2))_V^\theta$$

with two possible candidates G_1 and G_2 the choice splitting rule could be applied in two different ways. Depending on which candidate is selected, the order of resulting and-boxes will differ. This can of course lead to different results since the order of and-boxes makes a difference when the conditional guard operator is used.

One way to ensure confluence is to ensure that the candidate is selected in a deterministic way. This can be done only if the model provides a selection function from the sequence of goals in an and-box to a candidate.

The function alone does not help us if stability requirement is left out. If the function selects G_1 in the and-box above and the and-box can be rewritten the function might select G_2. The stability requirement ensures that the and-box is not rewritten.

The ordering of goals in an and-box lets us easily define a selection function (for example select the leftmost). If the candidate could be selected, predictable, in another way, the ordering of goals would be superfluous. This could possibly lead to a more efficient implementation and a more elegant language. The overhead for maintaining the order is however, as shown in Chapter 6, not very big.

4.6.3 Sufficiently stable

Does this mean that we are forced to wait for an and-box to become stable before we select a candidate and perform a choice splitting? No – we can perform choice splitting even in an unstable and-box without altering the outcome of the computation. It is for example safe to allow choice splitting

$$\mathbf{and}(S_1, \mathbf{choice}(\phi_X\ ?B_1, \sigma_Y\ ?B_2), S_2)_U^\theta\%E \overset{\chi}{\Rightarrow}$$
$$\mathbf{and}(S_1', \mathbf{choice}(\phi_{X'}'\ ?B_1'), S_2')_{U'}^{\theta'}\%E', \mathbf{and}(S_1, \mathbf{choice}(\sigma_Y\ ?B_2), S_2)_U^\theta\%E$$

if

- ϕ is quiet with respect to $env(\chi) \wedge \theta$ and X,
- σ is quiet with respect to $env(\chi) \wedge \theta$ and Y and
- no local goal in the sequence S_1 or S_2 can be subjected to a rewrite rule even if a compatible constraint is added to χ.

It does not matter if rewrite operations can be performed in the proper subtrees formed by goals in S_1 and S_2 as long as the local goals themselves are not rewritten. Why? - Only rewrite operations of the local goals in the and-box can add constraints to the and-box and only constraints added to the and-box can make ϕ_X or σ_Y fail. And this is the only way ϕ_X $?B_1$ can cease to be a candidate.

An and-box G is *sufficiently stable* in the configuration $\chi[G]$ iff no determinate rewrite rule is applicable in G that can add a constraint to the and-box even if constraints, compatible with the environment of χ, are added to χ.

This is interesting in an implementation since it allows us to be lazy in the execution of guards. If S_1 or S_2 contains a choice-box

$$\mathbf{choice}(\mathbf{and}()_U^\phi \rightarrow B, R)$$

we can ignore to do any rewrite operations in R as long as we know that ϕ is not quiet with respect to nor incompatible with $env(\chi) \wedge \theta$ and U.

If this is a wise thing to do, is another question. It depends on the cost of performing the computation of R twice, versus the cost of copying the result of the computation in R. If R can be solved after a large computation it is better to do the computation before doing the split operation. If the computation of R results in a large sub-configuration that is hard to copy it might be better to delay the computation. If the and-box $\mathbf{and}()_U^\phi$ becomes quiet after the split operation the computation of R might be avoided all together.

4.7 Non-determinism and deep guards

Apart from the non-determinism the computation model is simple. The non-determinism does create questions and the difficulties should not be ignored. Why do we have non-determinism in the language and are the deep guards really necessary.

4.7.1 Encapsulating search

Non-determinism in a language is useful since it allows a programmer to write programs that solve constraint problems in a compact and easily understandable form. Ask any Prolog programmer and he/she will tell you how easy a puzzle is solved. A common problem, however, in Prolog programs is that non-deterministic parts of a computations can interfere with parts that are deterministic. This is solved in the AKL computation model by encapsulating non-deterministic computations in deep guards. A goal with deep guards behaves like any other goal regardless if the guards contain non-deterministic computations.

One achieves a similar effect in a Prolog program if one always uses a cut in all clauses except when search was actually needed. The goals that are defined as search goals should then only be used in a `bagof/3` construct or encapsulated by a cut in another clause. AKL can then be seen as a language that encourages a good programming style ...oh sorry, I forgot, you are not allowed to use cut.

4.7.2 Reactive computations

The Penny system is targeted to supporting reactive computations. It has built-in procedures that use and/or modify the resources, such as files, provided by the underlying operating system. The built-ins which are connected to the operating system may only be used in the main and-box. Since non-determinism is encapsulated in deep guards the main and-box will never have to work with speculative computations i.e. it will never have to back-track.

The original AKL computation model allows choice splitting even of the uppermost and-box. If this is allowed we will have multiple reactive programs that all think that they have the right to manipulate the real world. To avoid this the Penny computation model was restricted to only allow choice splitting in sub-computations. The intuition is that the outside world is a process in the main and-box that always is able to make a determinate rewrite rule. The only restriction to the original AKL computation model is then that the uppermost and-box can not become stable and can therefore not be eligible for choice splitting.

4.7.3 Parallelism

Deep guards are not used as a mean to perform computations in parallel, but to encapsulate search. This is different from how the deep guards were used in the early committed choice languages for example Guarded Horn Clauses or Concurrent Prolog [59]. These languages used the deep guards as complex but deterministic tests and to spawn speculative parallel computations that could be pruned once a solution was found.

This sort of parallelism is in my experience hard to exploit and the programming style is not easy to follow. The deep guards did not justify the increased complexity of the implementation and were soon removed from the languages.

It is important to realize that the local constraint stores provided by the and-boxes are important even in a concurrent constraint programming language where no search is provided. In order to determine if a conjunction of constraints is entailed by, or incompatible with, an environment there is a need to, at least temporarily, hold a local constraint store.

If we are only looking for entailment each constraint could be examined separately. This is the case for committed choice languages, where the commit operator only is depending on the entailment of the guard. Several of these languages do however provide an “otherwise” constructor that makes the execution dependent also on the failure of guards. It is then highly desirable to have a clear understanding of what failure means. The concurrent constraint programming framework provides this semantics and an implementation can provide the correct functionality with a minimum of overhead.

Chapter 5

The Execution Model

BY AND EXECUTION MODEL we mean a refinement of the computation model, where more specific control principles are used. The execution model defines in what order rewrite rules are performed.

5.1 The execution model

The execution model is important as a tool for discussing different strategies without going into implementation details. The execution model is also important since it gives the programmer an operational view of what an execution in the Penny system is. The description given by the computation model does not answer questions of efficiency i.e. how big resources in terms of time and space an execution demands. The execution model gives a guideline to efficiency, although different implementations of the same execution model can have different behavior.

The execution model should be as free from implementation dependent decisions as possible, it is however important that the execution model is close enough to a real implementation so that one can estimate the complexity of the primitive operations.

The execution model is based on the execution model for the sequential AKL system that was used in the implementation of the AGENTS 1.0 system [34]. The sequential execution model was designed to favor non suspending computations in order to be competitive with non-concurrent languages. The model led to an efficient implementation that has, despite the complexity of the AKL computation model, only a small overhead compared to, for example, Prolog implementations.

The parallel execution model uses, as the sequential model, the notion of a *worker*. A worker is a process that performs rewrite operation on the configuration. The model is designed to allow several workers to operate on the same configuration. Each worker is driven by a set of *tasks*. The tasks are used not only to guide a worker through an execution but also as the entity for distribution of work among workers. The distribution of work is dynamic, it does not rely on a compile-time allocation of tasks to workers. A worker generates tasks during runtime that can be scheduled to be executed by another worker. The scheduling of tasks is a distributed process, it does not require any centralized resources.

The parallel execution model is designed for a shared-memory multiprocessor implementation. It is assumed that the configuration could be represented in a way that allows all workers to have equal access to the configuration. We also assume that the operating system can provide us with a set of processes that are executed fairly.

We use a fixed set of workers, the number of workers are determined by the implementation and not by the execution model nor the program being executed. This means that the execution model executes any program with the computational resources provided by the implementation. The model does not place any demands on the operating system in terms of more computational resources (apart from space and time).

5.2 The execution state

The *execution state* is a representation of the current configuration. The workers do not construct a whole new representation of the configuration in each rewrite rule, rather, they incrementally apply rewrite operations to the execution state to conform to the rewrite rules.

The execution state also includes a set of *dead* boxes. Boxes become dead when they are removed from the configuration. The dead boxes are not part of the configuration but need to be part of the execution state as long as there are valid references to the boxes. There might even be workers doing rewrite operations in dead boxes. This is a freedom we have since the result of rewrite operations in a dead box can never affect the execution in the configuration.

Boxes and expressions in the configuration are annotated to reflect their state. A guarded goal is when it is created marked as *untried*. When a worker starts to execute the guard it is marked as *tried*. And-boxes occurring in the configuration are *alive*, and-boxes that are removed from the configuration are dead. Dead and-boxes are either *failed*, *promoted* or *pruned*.

And-boxes and expressions are given *labels*, a label being anything which may serve as an identifier. Labels allow us to identify, informally, and-boxes and expressions in different, subsequent configurations. A suitable mental model is to think of a label as the address of a computer representation of an and-box or expression.

Conceptually, upon creation, each structure is given a label which is unique for the computation. Rewriting the interior of an and-box does not change its label. How rules preserve labels should be intuitive except maybe for the choice splitting rule which copies boxes. Each and-box and expression in the sub-tree formed by the left and-box is given a new label.

The variables and constraints that belongs to and and-box are *local* to the and-box and the constraints make up the *local environment* of the and-box. Variables and constraints in the path from the root to the and-box are *external* to the and-box and the constraints make up the *external environment* of the and-box.

5.3 Workers

A worker is located in an and-box or in a choice-box, the box is referred to as the *current box*. The current box is changed by *moving* between boxes. When moving to a box which is contained within the current box, the worker is said to move *down*. When moving to a box containing the current box, the worker is said to move *up* to a *parent box*.

A worker can always determine if it is alone in the sub-tree formed by an and-box. This knowledge is required to determine if an and-box is solved, stable or quiet. Since some information regarding the configuration is kept private to each worker, a worker cannot determine the state of an and-box only on the basis of its own knowledge. A worker can however only keep private information about the boxes in the path from the main and-box to its current and-box. A worker that is alone in a sub-tree therefore knows that no other worker can have private knowledge of the configuration represented by the sub-tree. It can thus determine properties such as whether an and-box is solved, stable or quiet.

5.3.1 Tasks

A worker uses *tasks* to guide its movements in, and operations on, the execution state. The tasks are associated to and-boxes in the execution state; each worker keeps track of its own tasks. There are be two kinds of tasks: *continuation* and *wake*. The tasks refer to, by the use of labels, and-boxes and expressions in the configuration for which rewrite operations should be tried.

A continuation task refers to an expression that has not yet been executed. The expression is either the original expression in a guard, or an expression added through the promotion rule. Their is, at any time, at most one continuation task per expression.

A wake task refers to an and-box that should be re-examined since either its external environment has changed, it is eligible for promotion or it is a possible candidate for choice splitting. The first case is the most common, the second and third cases occur only after a choice splitting; the worker starts to execute in the left copy but must be guided back to work in the right copy.

Associated to each variable is a set of wake tasks. The tasks should be considered when the variable is bound. When a binding is added to a variable X in an and-box A, the and-boxes of the associated tasks of X that are in the the sub-tree formed by

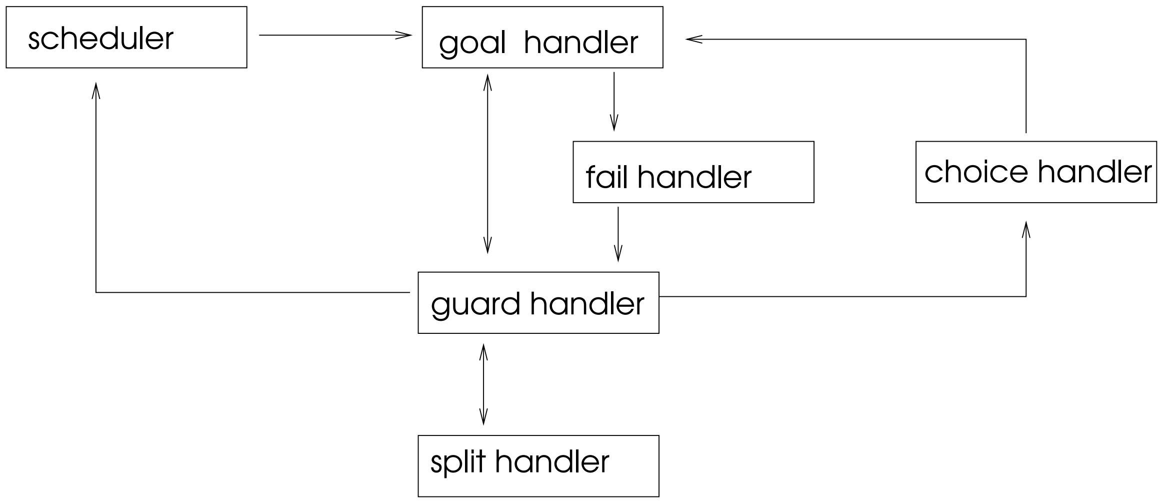

Figure 5.1: The execution model

A are called the *woken* and-boxes. The tasks of the woken and-boxes are removed from the variable and added to the set of tasks associated to the and-box. If X is an external variable a new wake task, that refers to the and-box A, is added to the variable.

Associating a wake task with a variable is called *suspending* it on the variable. The associated tasks are called *suspensions*.

5.3.2 Stacks

A worker associates tasks with and-boxes. Tasks are said to be *in* a box (from the point of view of a worker). A worker keeps track of its tasks, and which tasks are in which boxes. When a task has been processed, or when it is in a box which is removed because of failure, it is deleted. In this execution model, tasks are processed in a *last-in first-out* fashion. The worker processes all the tasks in a box before moving to a parent box. This allows the worker to keep its tasks in a stack.

5.4 The Handlers

The control is described through set of *handlers*. Each handler takes care of a particular sequences of operations. A worker performs the operations in a handler and either continue in the same handler or enters another handler. Each invocation of a handler is an undisturbed action, only one invocation is allowed at any moment, i.e. two invocations cannot occur in parallel. This is an unacceptable restriction in an implementation, but does not restrict the usefulness of the execution model.

Each handler is described in informal pseudo-code. Blocks, conditionals, and procedures are used, but natural language is used for most statements and expressions.

A handler is invoked by a `enter` command. A procedure in the handler is called with a `execute` command. We use natural language to describe the procedures and make use of implicit references to boxes to keep the language informal.

5.4.1 The goal handler

The goal handler is responsible for the execution of tasks. A worker first executes the continuation tasks. Only when all continuations have been handled are the wake tasks considered. When all tasks have been tried the worker moves to the guard handler.

```
the goal handler {
    if (there are continuation tasks in the and-box)
        remove and execute the first;
    else
        if (there are wake tasks in the and-box)
            remove and execute the first;
        else
            enter the guard handler;
}
```

A wake task is executed by first examining the woken and-box. If the woken and-box is dead the worker continues in the goal handler. If the and-box is alive the worker moves down to the and-box and re-examines the environment of the and-box. If they are incompatible the worker enters the fail handler, otherwise it enters the guard handler. Since no tasks can exist for the woken and-box, the worker can move directly to the guard handler without going through the goal handler.

```
execute a wake task{
    if (the and-box is dead)
        enter the goal handler;
    else
        move down to the and-box;
        if (environment is incompatible)
            enter the fail handler;
        else
            enter the guard handler;
}
```

The expression referred to by a continuation task is either a single atom or a composition. The continuation tasks are executed in a way that results in a left to right execution order of the atoms in a composition expression. Note that if the continuation task is the only reference to an expression containing more than one atom, then the created continuation task is the only task that refers to the remaining atoms.

```
execute a continuation task{
    if (the expression is an atom)
        execute the atom
    else
        remove the first atom from the expression
        add a continuation task for the remaining expression
        execute the atom
}
```

When a program atom is executed the worker first applies the program atom rule. The worker marks all but the leftmost guarded goal as untried and moves to the and-box of the leftmost guarded goal. After having added a continuation task for the guard expression it enters the goal handler.

```
execute a program atom {
    apply the program atom rule
    mark all but the leftmost guarded goal as untried
    move to the and-box of the leftmost guarded goal
    add a continuation task for the guard expression
    enter the goal handler
}
```

A constraint atom is executed by applying the constraint atom rule. If the constraint is satisfiable in the environment, new wake tasks are added for all of the woken and-boxes. If the environment is incompatible the environment failure rule is applicable and the worker moves to the fail handler.

```
execute a constraint atom {
    apply the constraint atom rule
    if (the local and external environments are incompatible)
        enter the fail handler
    else
        add wake tasks for woken and-boxes
        enter the goal handler
}
```

The requirement for moving to the fail handler can be relaxed. It is sufficient to determine if the added constraint is incompatible with the environment. This is possible since it does not matter if a worker adds a constraint to an already inconsistent environment and continues in the goal handler. If the environment already is inconsistent the and-box is removed from the configuration by the worker that added the inconsistent constraint.

The same relaxation can be made for the execution of a wake task. If the wake task not only refers to the and-box to be woken but also to a particular constraint it would be sufficient to determine if that constraint causes the environments to be incompatible. This is a freedom that is exploited in the abstract machine.

Any computation performed in a failed and-box is of course pointless but also harmless, it cannot interact with the remaining configuration. A worker that is executing in a dead and-box sooner or later checks, if it is not stuck in an endless computation, whether it is in a dead and-box or not.

Note that a worker can be in the goal handler even if the and-box has failed. The implementation can be made more efficient by removing a test for an exceptional case. There is no need to check if the current and-box has failed until the worker enters the guard handler. The performance of a single worker does not decrease.

Other atoms include aggregates, port related atoms and applications of abstractions. Port related atoms simply add their constraint to the constraint store of the current and-box. An apply atom or aggregate atom is executed in much the same way as a program atom.

5.4.2 The guard handler

The worker enters the guard handler when all tasks (of the worker) have been handled. The worker first determines if the and-box is the main and-box, if this is the case it enters the scheduler. If the and-box has failed, has been pruned (it cannot be in a promoted and-box) or if the worker is not alone in the and-box the worker moves up to the parent and-box and enters the goal handler.

```
the guard handler {
    if (the and-box is the main and-box)
        enter the scheduler
    else
        if (pruned, failed or not alone)
            move to the parent and-box
            enter the goal handler
        else
            execute the guard
}
```

There is an alternative approach and that is to let the worker move up to the parent choice-box and continue in the choice handler. This could be profitable since there might be untried guarded goals left under the choice-box. It is however perfectly safe to go directly to the parent and-box since the untried guarded goals are handled by the last worker that leaves or fail the and-box.

The guard is executed by the last worker in the and-box, it is only the last worker that can determine if the and-box is solved, quiet or stable. The worker applies, if possible, the pruning and/or the promotion rule. If a promotion is possible it moves to the parent and-box, adds the promoted constraints, creates a continuation task for the promoted body and enters the goal handler.

```
execute the guard {
    if (the pruning rule is applicable)
        apply the pruning rule
        mark pruned and-boxes as dead
    if (the promotion or unit rule is applicable)
        move up to the parent and-box
        apply the promotion or unit rule
        add wake tasks for woken and-boxes
        create a continuation task for the promoted body
        enter the goal handler
    else if (the collect rule is applicable)
        move up to the parent and-box
        apply the collect rule
        create a continuation task for the promoted body
        if (there are untried guarded goals left)
            move down to the choice-box
            enter the choice handler
        if (there is only one guarded goal left)
            move down to the and-box
            enter the guard handler
        enter the goal handler
    else if (the and-box is stable and a candidate is found)
        enter the split handler
    else
        move up to the parent choice-box
        enter the choice handler
}
```

The collect and unit rules are handled similarly to the promotion rule with one exception. When a collect rule is applied there might still be untried guarded goals of the parent choice-box. In this case the worker continues in the choice handler. The collection of the second-last guarded goal leads to that the unit rule is applicable. In this case the worker moves to the last guarded goal and enters the guard handler.

If neither a promotion nor a split operation is allowed the worker moves to the parent choice-box and enter the choice handler. But if the guarded goal has a conditional guard operator, the worker proceeds directly to the parent and-box and enters the goal handler. This can be done since it is possible to determine if the parent and-node is sufficiently stable even if the remaining guarded goals are not executed.

5.4.3 The choice handler

The guarded goals of the choice-box are examined. If there is an untried guarded goal the leftmost is selected. The worker moves to the and-box in the guarded goal, adds a continuation task for the expression (if any) in the and-box and enters the goal handler.

```
the choice handler {
    if (there are untried guarded goals)
        select the leftmost
        mark it as tried
        move to the and-box of the guarded goal
        add a continue task for the expression
        enter the goal handler
    else
        move up to the parent and-box
        enter the goal handler
}
```

If there are no more untried guarded goals the worker moves up to the parent and-box and enters the goal handler.

5.4.4 The fail handler

This handler is entered either if an incompatible constraint has been added to an and-box or if the last guarded goal in a choice-box has failed. If the and-box is already removed from the configuration, either pruned or failed by another worker, the worker moves to the parent and-box and continues in the goal handler. If however the removed and-box was the main and-box the worker terminates the execution.

If the and-box is dead because of failure the worker could move to the parent choice-box and look for untried guarded goals instead of moving to the parent and-box.

It is, however, safe to ignore these goals since the worker that actually failed the and-box has taken care of these goals.

An alternative approach is to let the first worker that detects a failure only mark the and-box as failed and move up to the parent and-box. It is then up to the last worker to leave the and-box to propagate the failure and take care of the remaining guarded goals.

If the and-box is still alive it is removed from the configuration. If the removed and-box is the main and-box the worker terminates the execution. Otherwise it examines the remaining guarded goals of the parent choice-box.

```
the fail handler {
    if (the and-box is dead)
        if (this is the main and-box)
            terminate execution
        else
            move up to the parent and-box
            enter the goal handler
    else
        apply either environment or goal failure rule
        move up to the parent choice-box
        apply the guard failure rule
        if (this is the root choice-box)
            terminate execution
        else if (there is only one guarded goal in the choice-box)
            move down to the and-box
            enter the guard handler
        else if (there are no guarded goals in the choice-box)
            move up to the parent and-box
            enter the fail handler
        else
            move up to the parent and-box
            enter the goal handler
}
```

If there is only one guarded goal left it might be eligible for promotion. This happens if the guarded goal has a wait or unit guard operator and is the last living guarded goal or if it has a conditional guard operator and is the leftmost guarded goal. The worker must then move to the guarded goal and resume execution in the guard handler.

If the guarded goal was the last one in the choice-box the worker must propagate failure to the parent and-box. Otherwise it moves to the parent and-box and enters the goal handler.

5.4.5 The split handler

The split handler applies the choice splitting rule with respect to a candidate (C). The parent choice-box of the candidate is called the fork (F). The parent and-box of the fork is called the original (A) and is the and-box that serves as a template in the construction of a copy (B). The parent choice-box of the original and-box is called the target (T) and its parent and-box the root (R).

The worker first moves to the root and-box. This means that it might move up one level or down several levels. It then applies the choice splitting rule. The right and-box is still referred to as the original, the left and-box is called the copy.

After the split rule has been applied the worker examines the original and-box. If the choice-box (F) that holds the remaining siblings of the candidate (C) is deterministic a wake task is created for the determinate and-box. Otherwise, if the original and-box (A) is stable, a wake task created for this and-box. The wake tasks are added to the and-box in order for the worker to find, return and apply either a promotion rule or a choice splitting rule. If these tasks are not added the worker can move up to the root and-box (R) and treat it as stable whereas in fact it is not.

```
the split handler {
    call the candidate C
    C's parent choice-box F
    F's parent and-box A
    A's parent choice-box T
    T's parent and-box R
    move up to the and-box R
    apply the choice splitting rule w.r.t. C
    call the copy of C, C'
    if (choice-box F is determinate)
        create a wake task for the and-box below F
    else if (the and-box A is stable)
        create a wake task for A
    move down to C'
    enter the guard handler
}
```

When this has been done the worker moves down to the copy of the candidate (C') and enters the guard handler.

Detecting stability

To detect stability one has to solve two problems: determine that no worker is positioned in the subtree formed by the and-box and determine that the subtree does not contain any non-entailed constraints on variables that are external to the and-box. There are several possible schemes for detecting stability that vary in precision.

One way to do this is to keep track of wake tasks. Associated with each and-box is information about wake tasks that are referring to the and-boxes within the and-box. When a suspension is added to a variable it is recorded in all ancestor and-boxes for which the variable is external. An and-box can then be determined to be stable if it no wake task has been recorded in the and-box.

When an and-box is woken a previously added constraint is re-examined. If the constraint turns out to be entailed all traces of the constraint must be removed from ancestor and-boxes. Even more problematic is the case when an and-box fails. Then the traces of all the constraints in the failed and-box and all and-boxes within it should be removed.

If either of these last two cases is not taken care of, boxes will be deemed to be unstable where they are not i.e. it cannot be established that they are stable.

The Penny system uses a precise scheme that does detect stability in all cases. This is important since computations otherwise could dead-lock. The AGENTS 1.0 system only partially detected stability, counting on the fact that the main and-box is trivially stable. This approach was not selected in the Penny system since a partial scheme would limit the potential parallelism.

Finding a candidate

In the execution model, the left-most innermost candidate is selected. This involves examining the leaves in the stable and-box left-to-right depth-first, choosing the first candidate G in an and-box $\mathbf{and}(S_1 \wedge \mathbf{choice}(G \vee R) \wedge S_2)^{\phi}_{U}$ where S_2 does not contain a choice-box $\mathbf{choice}(R_1 \vee \mathbf{and}(S)\%B \vee R_2)$ were a candidate can be found in S. By choosing the innermost candidate we reduce the size of the and-box that has to be copied.

5.4.6 The scheduler

A worker in the scheduler finds a new task by stealing one from another worker. If it can steal a task it continues in the goal handler.

A worker remains in the scheduler until a task is found or until all workers are in the scheduler. If all workers are in the scheduler then no rewrite rules are applicable and the execution terminates. This is the only time that a worker must be aware of which handler the other workers are executing in.

```
the scheduler {
    if (all workers are in the scheduler)
```

```
        terminate
    for (each of the workers not in the scheduler)
        if (the worker has a wake task)
            steal last task
            move down to the and-box
            enter the goal handler
    for (each of the workers not in the scheduler)
        if (the worker has a continuation task)
            steal last task
            move down to the and-box
            enter the goal handler
    enter the scheduler
}
```

When a task is stolen it must be removed from the owner and added to the stealer. The task is associated with an and-box or expression in the configuration and this association must not be lost in the process. Before the worker can enter the appropriate handler it must first move to the associated box.

5.5 Initial state

The initial execution state consists of a single and-box, with an empty set of constraints and an empty set of variables, containing a single expression. The and-box is referred to as the *main and-box* and the expression as the *main expression*.

In the initial state all workers are positioned in the main and-node. One worker has one continuation task. The task refers to the main expressions. No other worker has any task.

In the initial state all workers are in the goal handler. The worker that has the task for the main expression starts the execution. The remaining workers enters the scheduler. The execution terminates either when all workers are in the scheduler or when the main and-box has failed.

5.6 Properties of the Execution Model

Many alternatives exist in the design of the execution model. To gain certain advantages, others have to be sacrificed. It is important to discuss why the execution model has been designed in the way it is and the pros and cons of alternative approaches.

5.6.1 Environment stacking

A sequence of atoms is, through the continuation task, handled as one entity and is always accessed from the left to the right. This means that it is possible to represent the atoms by a single sequence of instructions i.e. in a compact form.

A worker in the scheduler will not steal part of a sequence but only the sequence as a whole. The scheduler preserves the sequence of atoms and not unnecessarily distribute the atoms over workers. The drawback is of course that a sequence of program atoms that preferably are executed in parallel has to pass through each worker that should contribute to the execution. Each worker starts to execute the first atom in the sequence and lets the remaining sequence be stolen.

The alternative is to use what is often called goal stacking i.e. the first one to execute a sequence breaks it up into goals that then could be shared among workers. The drawback with this approach is that it induces an overhead for the sequential left to right execution.

5.6.2 Conservative

The execution model is conservative when it comes to exploiting parallelism in different guarded goals under a choice-box. The worker that creates a choice-box starts to work in the leftmost guarded goal but there is no mechanism for other workers to start working in the remaining guarded goals. Parallel execution in guarded goals only occur if:

- the first guarded goal is suspended and later woken by another worker or
- choice splitting is performed and another worker steals the task that is referring to the stable or deterministic and-box in the right branch.

The conservative approach can limit the amount of parallelism in an execution but there are pragmatic reasons to select this strategy. The exploited parallelism is only limited if no other work is available and it is then only severely limited if the granularity of the work in the guarded goals is large enough to justify the parallel execution.

To determine how serious the limitation is to performance it is important to look on the structure of AKL programs. An analysis of many AKL programs would of course be preferred but since only a small number of "real" AKL programs have been written I can only use my own experience as a basis for my conclusions.

To my experience the use of deep guards i.e. guards containing program atoms is limited. It is rare that two or more guards of a definition are deep. The main usage of deep computations is to encapsulate a non-deterministic computation and in this case the conservative approach does not limit the amount of available work.

The reason for being conservative is that program atoms are executed before all information is available more often in the parallel system than in a sequential system. This is a consequence of the implicit parallel execution model and means that the parallel system creates more suspended guarded goals. These guarded goals must then be woken, a process that is expensive. This can lead to serious deficiencies especially for conditional guarded goal. Consider a definition with a sequence of conditional guarded goals that all suspend on a variable. When the variable is bound the worker has no knowledge of which of the wake tasks that pertains to the leftmost guarded goal. If the guarded goals are woken in a right to left order all of the guarded goals must be woken before a promotion is allowed. This is clearly undesirable but the conservative approach solves the problem at cost of loosing potential parallelism.

The execution model could be even more conservative. Consider a nondeterministic computation that is encapsulated in a conditional guard. Only the leftmost solution is promoted and any execution in guarded goals that are to the right of the solution is pointless. To execute the guarded goals in parallel is speculative in the same way as or-parallel Prolog executions can be speculative. In an or-parallel Prolog system it is important to minimize the speculative work by trying to move the workers to the left in the search tree. The same approach would be applicable in the Penny system but I argue that the whole issue could be solved by never doing any parallel executions at all in conditional guards.

This does of course look like I'm hiding my head in the sand but there are good reasons to support such a strategy. In a concurrent constraint programming system there is often available work in the reactive top level. If this is the case there is no need to consider parallel work in the conditional guarded goals. If this is not the case it means that the main part of the execution consists of one problem that is solved using the Penny non-deterministic capability. It is of course easy to write a toy benchmark with these characteristics but in a real application one would use some problem specific search mechanism instead of relying on the default mechanism provided by the Penny system.

In AKL and Penny the split operation is hard-wired into the computation model. A more powerful approach is to handle the split operation at the program level. A higher order combinator, $solve(A, Z)$, connects an abstraction A with the result Z. The result will be equal to either:

fail if A can be rewritten to a failing and-box,

solved(S) where S is an abstraction that when applied returns the only solution to A or

split(L,R) where L and R are two abstractions, containing two possible solutions to A.

The split operation is only performed if the abstraction A evaluates to a stable and-box. If this is the only way a split operation is performed, stability must only be detected in and-boxes that are created by the solve combinator. This can lead to simplifications in the implementation that we discuss later.

The higher order approach to handle search was developed in the DFKI-Oz programming system and has later been included in the AGENTS system [57]. Is is powerful since it gives the programmer the tool for implementing a variety of search strategies.

Note that the question of how parallelism should be exploited now is under the programmer's control. The programmer may wish to execute both L and R concurrently or let the execution of R wait until the execution of L has given a result.

5.6.3 Eager or lazy

The execution model is lazy when it comes to handling wake tasks. The wake tasks are handled only after all continuation tasks have been executed. This has some drawbacks since the configuration could become unnecessary large. If an and-box is suspended on a variable it would be natural to re-examine the and-box as soon as the variable is bound. The hope is that the and-box can be promoted and add new bindings to the constrain store and thus prevent other and-boxes from suspending. Why expand the configuration by executing expressions when we might remove and-boxes in wake operations?

When things break down

The obvious benefit of an eager execution model is however deceiving in an implicit parallel system. This can be shown with a small example using the `qsort/3` definition listed in Figure 5.6.3 and a goal `qsort(List, [], Sorted)`.

Using one worker the execution is done, if `List` is bound to a list of N elements, using $log(N)$ continuation tasks and a fixed number of living boxes. If `List` is bound to a single cons cell the first call to `partition/4` results in a choice-box with suspended guarded goals. The worker then handles the continuation task and execute the two `qsort/3` atoms, both results in choice-boxes with suspended guarded goals.

If the elements of `List` are produced one after the other, and the suspended goals of `partition/4` and `qsort/3` always are given a chance to proceed as soon as they can, the resulting configuration has N suspended choice-boxes for `partition/4` and N suspended choice-boxes for `qsort/3`. This is of course unacceptable.

One can of-course argue that the normal way of writing programs is in a left-to-right order i.e. in a way that would ensure that data always is present when an atom is executed. This is right but it does not save us in system with implicit parallelism.

If two workers are used, and `List` is bound to a list of N elements, the first worker starts to execute the first `partition/4` atom. The second worker steals the con-

```
qsort([], S0, S):-
    ->  S0 = S.
qsort([H|T], S0, S):-
    ->  partition(T, H, Low, Hi),
        qsort(Low, S0, [H|S1]),
        qsort(Hi, S1, S).

partition([], _, Low, Hi):-
    ->  Low = [],
        Hi = [].
partition([X|T], P, Low, Hi):-
        X < P
    ->  Low = [X|L],
        partition(T, P, L, Hi):-
partition([X|T], P, Low, Hi):-
    ->  Hi = [X|H],
        partition(T, P, Low, H)
```

Figure 5.2: The quick sort program

tinuation task and executes the following `qsort/3` atoms. Since the list `Low` is consumed (in `qsort/3`) faster than it is produced (only about half of the elements of `List` are added to `Low`) the execution results in a number of suspended goals. When the first worker adds an element to the list `Low` it is notified that a suspended goal should be woken. If we have an eager strategy it preempts its current execution of `partition/4` and moves to the suspended goal. This only leads to that the production of elements of the lists `Low` and `Hi` is further delayed.

The second worker steals, after having suspended its execution, the continuation of the first worker and resume the execution of the `partition/4` procedure. It is only able to produce one additional element before it wakes a suspended goal.

The execution results in the same explosion of suspended goals as if one worker was used and the elements of `List` arrived in a slower pace than they could be consumed.

When does it work

Is it then always wrong to have an eager wake execution model? – No, if the woken goal can be reduced or at least reduced to something that is smaller it might be useful. Allowing a consumer to run will hopefully also decrease the amount of live data in the execution. In general it is however hard to tell if a procedure is a net consumer or producer of data.

```
qsort([], S0, S):-
        |       S0 = S.
qsort([H|T], S0, S0):-
        |       qsort(Low, S0, [H|S1]),
                qsort(Hi, S1, S),
                partition(T, H, Low, Hi).
```

Figure 5.3: Never do this in an eager system

An eager execution model is used in the KLIC system. If the `qsort/3` procedure listed in Figure 5.6.3 is executed the system will behave badly.

The system will suspend the two `qsort/3` goals and then execute the `partition/4` goal only to go back to one of the woken `qsort/3` goals. The execution time will be a magnitude higher compared to the normal `qsort/3` definition.

No one would of course define `qsort/3` in this way but how does the system work if more workers are used? The model works in KLIC since there is no implicit parallelism; goals that should run in parallel are explicitly annotated with nodes.

If the `partition/4` and `qsort/3` procedures were executed on their own nodes, the suspensions generated by the `qsort/3` procedures will not cause the execution of the `partition/4` procedure to terminate. The node that is assigned the `partition/4` procedure will only notify the nodes executing the `qsort/3` procedures that goals have been woken, It will not move to these goals and do the wake operation itself. The production of elements for `Low` and `Hi` can thus proceed as fast as possible (although there is an overhead of notifying nodes of woken goals).

The KLIC system also has additional support to control parallel activities. Goals can be annotated with different priorities. This can be use to give producer (the partition procedure) higher priority than the consumers (the qsort procedures).

The Penny way

In the Penny system we have chosen to let the goal handler work in a lazy way and let the global scheduler work eagerly. For a worker in the scheduler it is indifferent if it steals a wake or a continuation task but stealing a wake task is less disturbing for the busy worker.

The scheduler will also steal the last task i.e. the first wake or continuation task generated by the busy worker. If it is a wake task and that wake task is referring to a suspended goal such as the `partition/4` procedure in the quick-sort example, it is likely that this procedure has more than just one datum available. The extreme ping-pong effect of eager wakening is thus to some degree avoided.

The problem could of course be avoided by restricting the concurrency in the language. The original DFKI Oz system defines the execution order to be "left to right" as long as no goals are suspended [64]. A parallel implementation would however have reduced chances of exploiting parallelism since it would have to make sure that the sequential execution order is maintained.

5.6.4 Fairness

The execution model is in no way fair. A worker can easily be stuck in an endless computation and there is no guarantee that tasks eventually will be executed. A fair execution model is harder to define. A worker must not only process the task in an and-box in a first-in-first-out order but process all task in the configuration in a that order. A worker would then have to keep track of not only tasks in the path from the root to the current box but in all parts of the tree.

One way of describing fairness would be to create workers dynamically and to state that the workers are scheduled fair by some underlying machinery. This is not a solution of the problem but rather a way of ignoring the problem.

5.6.5 Concurrency

The implicit concurrency is important for problem solving but is not needed in a reactive computation. We could explicitly divide the expressions in an and-box into *threads*. The atom rules would only be allowed to operate on the leftmost atom in a thread and new threads could be introduced by an additional rewrite rule.

Threads would make a declarative semantics of the language harder to formalize but would make the operational semantics much more useful. One would be able to predict the size of the configuration something that, as far as the semantics goes, is irrelevant.

Chapter 6

The Abstract Machine

An abstract machine is defined that allows workers to build and manipulate a shared execution state. This chapter describes the representation of the execution state, the locking scheme and the instructions of the abstract machine.

6.1 Why an abstract machine

An abstract machine is a machine that is specified using abstract entities such as stacks, flags, registers, memory etc without describing how these should be implemented. Abstract machines have been used to give clear operational semantics to as well as describing the execution mechanism of languages languages [40]. In the logic programming community the semantical properties of languages have been described by model theory or rewrite systems. The abstract machines have served mainly as tool for language implementors.

An abstract machine is often implemented in software although several machines have been at least partly implemented in hardware. The abstract machine presented in this chapter has borrowed many concepts from the Warren Abstract Machine (WAM) [72, 2].

6.2 The execution state

The execution state is divided into: a representation of the configuration which is shared between all workers, the structures that are local to each worker and the representation of the terms. All parts of the execution state are equally accessible to all workers. Even the local structures of a worker are accessed by other workers but they are primarily managed by the owner.

The abstract machine is designed to allow multiple workers to collaborate in parallel. This means that some of the data structures that are used in the representation of the execution state must be protected by locks to guarantee a consistent representation.

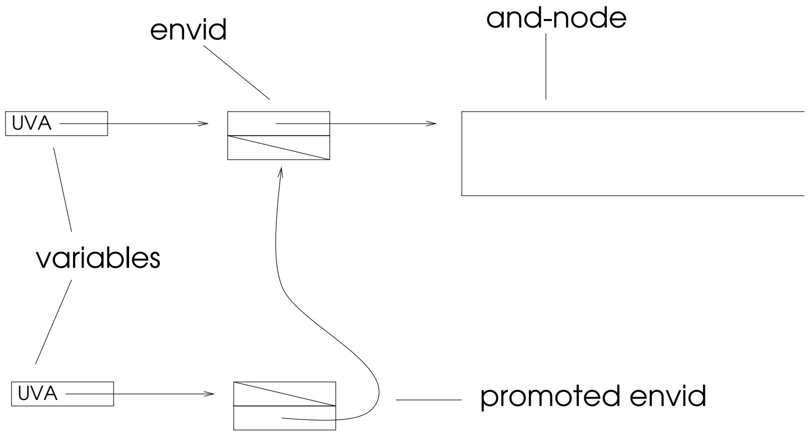

Figure 6.1: The representation the environment identifier

Very little is assumed about the locks. Workers take locks but only one worker at the time can hold a lock and a lock is held the until the worker wishes to release the lock. A lock does not have to be starvation free. A worker cannot do much work if a lock cannot be taken but as long as other workers are working the whole computation makes progress. There are few cases when a worker needs to hold two locks, in these cases an ordering is always defined that prevents deadlock.

6.2.1 The configuration

The configuration is represented as a tree of *choice-nodes*, *and-nodes* and *and-continuations*. The root of the tree is a dummy choice-node that serves only as a sentinel in the execution. The root node holds a single and-node that is referred to as the *main and-node*. In the following sections we refer to the parent, the children and the sibling of nodes.

And-node

An and-node is the representation of an and-box. It holds a sequence of choice-nodes and and-continuations, that represents the goals in the guard, and a set of *constraint entries*. Since there is a one-to-one relation between and-boxes and guarded goals, a guarded goal is represented by an and-node holding the guard operator and a pointer to a body. The body is represented by an and-continuation. The main and-node does not hold a body nor a proper guard operator.

Some of the operations on the and-node must be protected by a lock so each and-node is equipped with a lock. The lock is placed in a separate structure called the *environment identifier* (envid) of the and-node. The environment identifier is separated from the and-node to allow us to reclaim the and-node when it is removed from the configuration.

The environment identifier contains a three-value lock and a forward pointer. The lock takes the values: and-node pointer, locked or dead. The and-node pointer is a pointer to the and-node to which the environment identifier belongs. The forward field is either null or, if the and-node has been promoted, a pointer to the environment identifier of the parent and-node. It is possible to code both the lock and the forward pointer in one word by tagging the different pointers but the two word representation has been kept in the Penny system to facilitate debugging.

The set of variables of an and-box are not accessible from the and-node. In order to determine the home of a variable all variables keep a pointer to the environment identifier of their home and-node. If the and-node is promoted the forward pointer is dereferenced until a living environment identifier is found. The picture in Figure 6.1 shows two variables that both belong to an and-node. The upper variable was created in the and-node and has an environment pointer that points directly to the environment identifier of the and-node. The lower variable belonged to an and-node that has been promoted. As the and-node was promoted the environment forward-pointer was set to the environment identifier of the parent and-node.

Goals

The sequence of goals must be ordered in order for the split operation to find the leftmost candidate. If this was not required by the language the operations on the sequence are simplified but not necessary more efficient. The abstract machine is designed in a way that allows explicit reuse of goal structures. For this reason the goals must be accessible from the and-node. The ordering requirement is not hard to implement and actually provides for operations on the sequence to be performed in parallel.

The sequence of goals in the guard is represented by a single linked list of *insertion cells*. Each insertion cell is tagged and is either *dead*, *free*, pointing to a *choice-node* or pointing to an *and-continuation*. The choice-nodes and and-continuations also holds pointers to their insertion cells. To facilitate insertion the cells are linked right to left, i.e. the leftmost goal in the guard is the last goal in the list.

A new cell is always inserted to the left of an and-continuation. A worker that needs to insert a new cell does this simply be linking the cell of the and-continuation to a new cell and let the new cell point to the next cell in the list. This operation can be done in parallel and without any locking. Figure 6.2 shows the representation of the list of goals.

The leftmost goal cannot be found unless the whole list is traversed. This might seem to be a deficiency but since the leftmost goal is only located when choice splitting is performed the overhead is small. An alternative would be to keep a pointer to the leftmost goal but this would induce an overhead in many operations even if no choice splitting is performed.

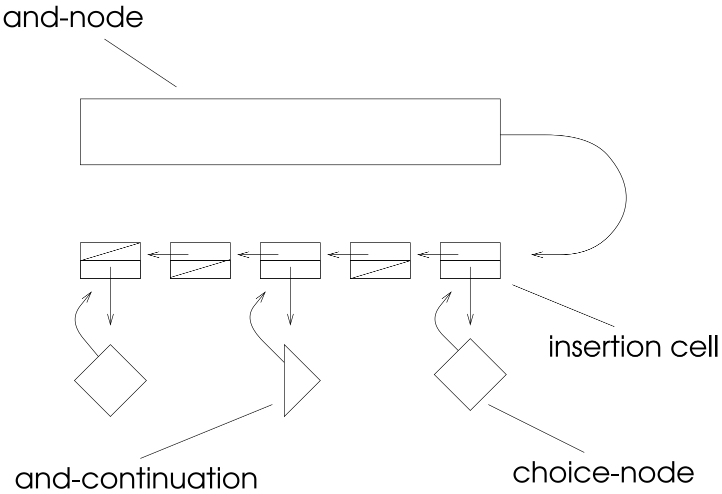

Figure 6.2: The list of goals in an and-node

The advantage of the scheme is that no locking is necessary in order to keep the list ordered. A disadvantage is that the list of insertion cells grows and that it contains dead entries. Dead entries can of course be removed at any point in the execution, during execution or during garbage collection, but the list must be traversed by a worker if it wishes to determine if the and-node is solved. The scheme therefore allows dead cells to be reused so that the number of cells are kept to a minimum. The operations on the list are described in the following sections.

Constraint entries

The set of constraint entries are used to keep track of local bindings of external variables and of suspended, determinate and stable and-boxes. Entries of an and-node are also structures that we want to reclaim explicitly. We therefore protect each entry with a *hanger*. The hanger contains a pointer to the environment identifier of the and-node and a pointer to the entry itself. Figure 6.3 shows the representation of an and-node with a single entry.

The pointer to the environment identifier also serves as a lock of the entry. A worker that wants to access the entry must first take the lock. Until the lock is held nothing should be assumed about the entry. Once the lock is taken the first operation must be to check whether the entry is still alive. If the entry is dead the lock should be immediately released to allow other workers to take the lock.

A constraint entry is either a *unifier entry*, a *suspension entry* or a *continuation entry*. A unifier entry holds a reference to a variable and a term to which the variable is locally bound. A suspension entry holds a reference to a variable and a set of *suspensions*. Each suspension is referring to an entry that belongs to an and-node on a level immediately below the and-node of the suspension entry. Suspensions are explained in Chapter 7.

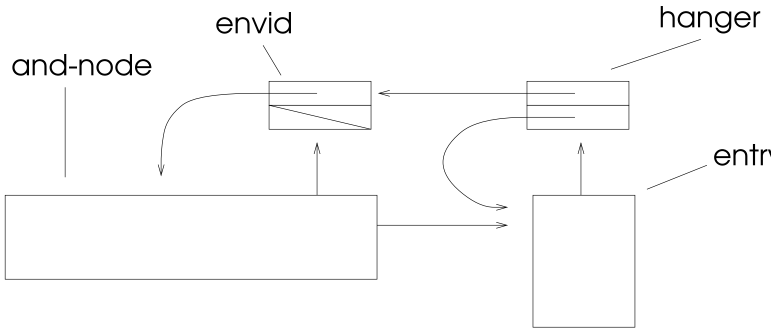

Figure 6.3: The representation of an entries

A continuation entry holds a possibly empty list of suspensions. It is similar to a suspension entry but it is not associated with a variable. An empty list is used to indicate that the and-node is stable or eligible for promotion. This is only used in conjunction with choice splitting and is further described in section 6.3.6.

State

The and-node also has a type that is either one of the guard operators: *wait*, *conditional*, *commit*, *collect*, *unit* or one of *pruned*, *failed*, *main* or *error*. The error type is used for exceptional cases such as failure of the main and-node.

A worker must be able to determine if it is alone in the and-node. A bitmap is used to record the workers that are active in or below an and-node. A worker that wishes to work in an and-node must first install itself in the and-node. To install itself the worker must first take the lock of the and-node (in the environment identifier) and then add a unique bit to the bitmap.

A worker that is not installed in an and-node is not allowed to modify the node, nor trust any previous knowledge of the node. The only operation that is allowed is the lock operation. To determine if a worker is alone in an and-node the lock is taken, if the worker is the only registered worker it is alone.

And-continuation

An and-continuation represents a sequence of atoms by a tuple of registers and a program counter. The program counter is a pointer to a sequence of instructions. An and-continuation is similar, in its structure and use, to an “environment” in WAM [72].

The first and-continuation allocated in an and-node implicitly holds the representation of both the guard and body of the guarded goal. The body need not be explicitly represented since it is only needed after the instructions that pertain to the atoms of the guard have been executed. Once the instructions of the guard have been executed then the and-continuation represent the body of the guarded goal. The body is used if the and-node is promoted.

The and-continuation has no lock. The abstract machine guarantees that only one worker has access to the and-continuation at any given moment.

Choice-node

A choice-node represents a choice-box. It contains a sequence of and-nodes and one *choice-continuation*. The choice-continuation represents a sequence of untried guarded goals and is similar to a choice-point in WAM, it holds a program pointer and a copy of the *argument registers*. The choice-node does serve a purpose even when all alternatives have been tried or when there is only one clause to start with. This is different from a choice-point. A choice-point is only needed as long as there are untried alternatives left. The choice-continuation is replaced by a null-pointer if all guarded goals have been tried.

The operations on the list of and-nodes is different from the operations on the list of goals. A worker must be able to remove (by pruning or failure) any and-node and this requires some locking. A fast insertion operation, provided in the list of goals, is not essential. New and-nodes are only created from a single choice-continuation or as the result of a split operation. Both operations are infrequent.

The and-nodes are linked in the list starting with the rightmost and-node. The list is linked from right to left in order to facilitate the insertion of a new and-node in the *retry* and *trust* instructions. A lock in the choice-node is used to synchronize modifications of the list of and-nodes and choice-continuation.

The choice-continuation allows a lazy creation of and-nodes. A choice-node without any and-nodes but only a choice-continuation can be seen as a suspended program atom. All information needed to apply the program atom rule is contained in the choice-node and its continuation.

A dummy choice-node is used as the root of the tree. This node is necessary in order for every and-node to have a parent choice-node. This fact is used in the fail handler to avoid an unnecessary test.

6.2.2 The worker

A worker is either active and positioned in an and-node in the configuration, or idle. The node in which the worker is positioned is called the current and-node. We also refer to the *current and-continuation* and the *current insertion cell*.

Stacks

The worker keeps four stacks: *continuation stack*, *recall stack*, *wake stack* and *context stack*.

continuation Each continuation task is a pointer to an and-continuation that represents an untried sequence of atoms.

wake Each wake task is a pointer to a constraint entry. In the execution model a wake task was used to guide the execution to an and-box that should be re-examined. The wake task in the abstract machine indicates exactly which constraint that should be re-examined.

recall Each recall task is a pointer to a suspended goal represented by a choice-node. The recall task is a more efficient form of a wake task.

context The context stack divides the task stacks into segments, each segment corresponds to an and-node in the path from the root to, and including, the current and-node. Each entry on the context stack therefore holds pointers to the top of the task stacks as they appeared when the and-node was entered. Each context also holds a reference to the and-node to which they belong.

The task stacks could implicitly be divided into segments by using sentinel tasks. The fail handler requires that all tasks of the failing and-node are removed. It is therefore convenient to have this information explicitly recorded on the context stack. If a sentinel task was used the stacks would have to be scanned in search for the first sentinel. The advantage of having a sentinel is that the stack boundaries do not have to be checked when tasks are popped from the stacks.

Registers

The instruction handler needs some registers: a *code pointer*, a *structure pointer*, a *mode flag* and a *set of argument registers*. The code pointer, structure pointer, mode flag and argument registers are use much in the same way as in WAM[72]. The worker also has registers to hold pointers to the current and-box, current and-continuation and current insertion cell.

These registers are only valid within the instruction handler and need not be saved when the worker exit from the handler.

6.2.3 The terms

There are three kinds of terms: variables, structures and atoms. A variable holds a reference to the environment identifier of its home and-node and a list of suspensions. A structure holds a functor (the name and arity) and a tuple of terms. Atoms only have an identity.

There is of course a lot more that should be said about both the properties and representation of different terms but for the understanding of the abstract machine this description is sufficient for now. The exact representation of terms is discussed in Chapter 7

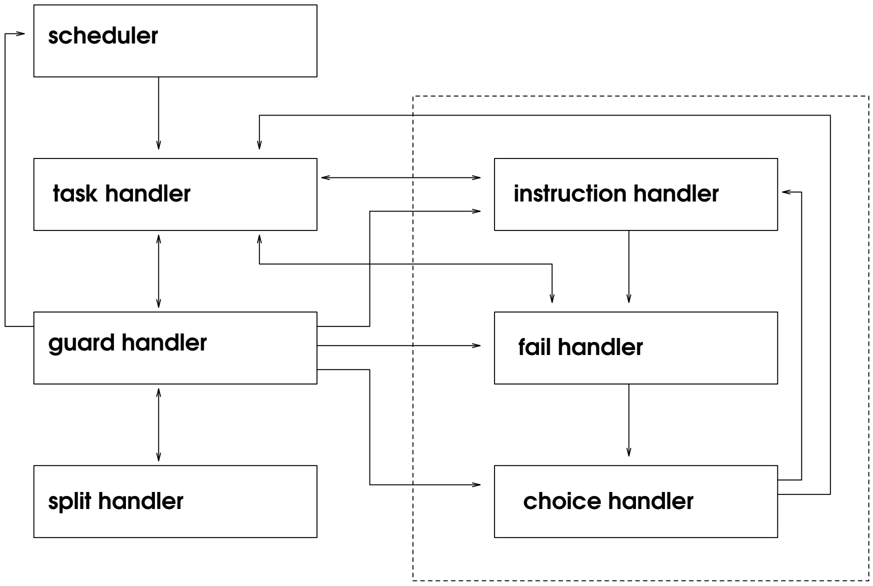

Figure 6.4: Flow of control in the abstract machine

6.3 The handlers

In the description of the abstract machine, the goal handler has been divided into a *task handler* and an *instruction handler*. The instruction handler takes care of the execution of program atoms and constraint atoms and the task handler takes care of the and-box tasks. The other handers are, as described by the execution model: *fail handler*, *choice handler*, *guard handler*, *split handler* and the *scheduler*.

Figure 6.4 illustrates the flow of control between the handlers. The instruction handler can together with the fail handler and choice handler be seen as the WAM part of the abstract machine. Note that this is only one part in the abstract machine. This part is of course important to implement as efficiently as possible using compilation techniques as well as implementation techniques but our work is focused on the implementation of and interaction between the other handlers since these are unique to our system.

6.3.1 Instruction handler

The instruction handler is responsible for calls, selection of guarded goals, building of data structures and open coded unification.

The handler is entered through the task handler, when an untried goal is executed, or through the choice handler, when an alternative guard is tried. A worker leaves the instruction handler and enters the fail handler if a call of a goal or a unification fails. If a proceed or suspend instruction is executed the worker leaves the instruction handler and enters the task handler.

During an invocation of the instruction handler the worker can generate both continuation, wake and recall tasks. The wake and recall tasks are created when constraint atoms are executed. A worker can while in the instruction handler build and-nodes and choice-nodes. It can build the nodes to any depth but can only enter the nodes, never leave them.

The instruction handler is described in detail when the instructions are presented.

6.3.2 Task handler

The tasks are the primitive operators of the concurrent computation. There are three types of tasks: continue, wake and recall. Continue tasks are generated by the call instruction, wake and recall tasks are generated when a variable with suspensions is bound.

Since the different types of tasks are stored on separated stacks the tasks can be handled type by type. The worker first examines its continuation task stack and then the recall and wake stacks. The first task is selected and removed from the stack. During the operation the stack lock must taken, this is done in order to prevent other workers from stealing the task.

Continuation

The continuation task refers to an and-continuation in the current and-node. The and-continuation is made the current and-continuation. The worker enters the instruction handler starting with the instruction pointed to by the continuation program-pointer of the and-continuation.

Recall

A recall task refers to a choice-node immediately below the current and-node. This is the only recall task that is pointing to the node so no locks have to be taken in order to modify the node. The saved argument registers, found in the choice-continuation, are copied back to the argument registers of the worker. The choice-node is removed and the insertion cell of the node is made the current insertion cell. The worker then continues in the instruction handler starting with the first instruction of the choice-continuation.

Wake

A wake task refers to a constraint entry of an and-node immediately below the current and-node (or rather below a choice-node below the current and-node). If the lock of the entry is taken the worker installs itself in the and-node, pushes a new context and removes and examines the entry.

The constraint entry can, as said before, be of three different kinds: unifier, suspension or continuation.

unify The two arguments are unified If the unification fails, the worker enters the fail handler. If the unification succeeds new recall and wake tasks might have been generated. The worker therefore continues in the task handler.

suspension The suspension list is transformed to tasks that are added to the appropriate task stacks. The worker then continues in the task handler.

continuation If the entry is empty the worker can continue in the guard handler. If it contains a list of suspension, these are transformed to wake tasks and the worker continues in the task handler.

When all task stacks are empty the worker continues in the guard handler.

6.3.3 Guard handler

The worker enters the guard handler after all tasks associated with the and-node are processed. In the guard handler it checks the type of the and-node. If the type is *main* it enters the scheduler, if it is *error* it terminates the execution. If the type is a regular guard operator it locks the parent choice-node and current and-node (in this order). The worker then determines if it is alone in the and-node. If it is not alone it de-installs itself from the and-node, releases the locks of the and-node and choice-node and enters the task handler.

If the worker is alone in the and-node it examines the type of the and-node. The type can be either: failed, pruned, conditional, commit, wait, collect or unit. The following actions are performed:

failed A failed type means that another worker has failed the and-node but has left to the last worker to actually perform the guard failure operation. The worker enters the second phase of the fail handler in a state where it is already determined that the worker is alone in the and-node and that the and-node and choice-node are locked.

pruned A pruned type means that another worker has pruned the and-node. The worker releases the locks of both nodes, moves up to the parent and-node and enters the task handler.

conditional The worker first checks if pruning is possible. This is done by examining if the and-node is solved and quiet. After having pruned sibling and-nodes (to the right) the possibilities for a promotion is determined. If a promotion is possible the worker marks the node as promoted, releases the lock and moves up to the parent and-node. The body of the promoted and-node inherits the insertion point of the removed choice-node, the program counter is updated and the worker continues in the instruction handler.

commit The commit operation is similar to the conditional operation. The only difference is that the commit operation prunes in both directions.

wait The wait operation is different in that pruning is not allowed and promotion is more difficult. If promotion is performed the bindings of the and-node are added to the parent and-node. This means that the same precautions must be taken as when a constraint atom is executed. If a binding fails the worker enters the fail handler.

collect The collect operation is similar to the conditional operation. The difference is that the collect operation does not prune sibling and-nodes nor does it remove the choice-node if the and-node is promoted. A new insertion cell is created to the left of the choice-node and a continuation task is added for the promoted body. If the and-node is collected and the choice-box has only one remaining and-node the worker moves to that and-node and enters the guard handler.

unit The unit operation is similar to the promotion of a single and-node in a conditional operation.

In a pruning operation the choice-continuation of the parent choice-node is removed and pruned sibling and-nodes are marked as pruned. Note that a worker can never find itself to be in a promoted and-node. The requirement for promotion is that the promoting worker is alone in the and-node. The worker holds the lock of the and-node and prevents any worker form entering until the and-node is marked as promoted. A worker never enters a promoted and-node.

If a promotion is not allowed the worker must examine if a split operation can be performed. If the and-node is stable and a candidate is found the worker enters to the split handler.

If a split operation is not possible, the worker is done. It then releases the lock of the and-node. The simplest thing would now be to move to the parent choice-node and enter the choice handler but we can take a shortcut. If the and-node represents a guarded goal with a conditional guard operator we know that no promotion is possible unless the current and-node fails or is promoted itself. We can therefore delay the creation of the sibling and-nodes. By moving directly to the parent and-node and enter the task handler we postpone the creation of the sibling and-nodes.

6.3.4 Choice handler

In the choice handler the worker looks for a choice-continuation. If a choice-continuation exists, the saved registers are copied back to the argument registers. The worker then enters the instruction handler starting with the instruction pointed to by the continuation program pointer. If no choice-continuation exists, the worker releases the lock of the choice-node, moves to the parent and-node and enters the task-handler.

6.3.5 Fail handler

The fail handler is entered from the instruction handler, guard handler or task handler. If the worker enters from the instruction handler or task hander it first takes the locks of the parent choice-node and current and-node (in this order). Once the lock of the current and-node is taken the workers determines if it is alone in the and-node.

If the worker is not alone in the and-node the actions are determined by the and-node type in the following way:

fail, pruned, conditional, commit, wait or collect The worker marks the and-node as failed (if not already), moves to the parent choice-node and enters the choice handler.

main The worker marks the and-node with error and terminates the execution.

error The worker terminates the execution.

If the worker is alone in the and-node the actions are determined by the and-node type in the following way:

pruned The worker moves to the parent and-node and enters the task handler.

failed, conditional, commit, wait or collect The worker marks the and-node as failed (if not already) and continues in the second phase of the fail handler.

main The worker marks the and-node with error and terminates the execution.

error The worker terminates the execution.

The usage of the error type is important in order to signal to other workers that the and-node is the main and-node. If the error type was not used a worker would have to examine each failed and-node to see if the and-node is the main and-node.

When the worker moves up to the parent and-node or terminates the execution it must of course first de-install itself from the current and-node and release the locks of both the and-node and the parent choice-node. If the worker proceeds to the second phase of the fail handler it keeps both of the locks.

The second phase of the fail handler takes care of the guard failure rule. This phase could also be entered from the guard handler directly when a worker detects that it is the last worker in an already failed and-node.

The worker removes the node and checks if the node was the last and-node of the parent choice-node. If this was the case the worker removes the parent choice-node, enters the parent and-node and again invokes the fail handler. If only one sibling and-node exists this and-node could be promoted. The worker therefore adopts the node as the current and-node and proceeds in the task handler. If none of the special cases apply the worker moves up to the choice-node and enters the choice handler.

6.3.6 Split handler

When the worker enters the split handler, it holds the lock of the current and-node and its parent choice-node. The worker first positions itself in the parent and-node of the candidate and-node, called the original node (this might mean that the worker moves down in the tree but the original node often turns out to be identical with the current and-node). It then ensures that the original node and its parent choice-node are locked and then inserts a new locked and-node, called the copy, to the left of the original node. Once the copy is inserted, the lock of the parent choice-node can be released to allow other workers to make split operations in sibling nodes.

After having copied the content of the original node into the copy, it examines the original node. If the candidate had only one sibling and-node this node is deterministic and can, if it is solved, be promoted. To find this task, an empty continuation entry is added to the sibling node. This entry is then referenced by a continuation entry in the original node. If the candidate had more than one sibling but the original node is stable an empty continuation entry is added to the original node. If an entry is added to the original node, a wake task for the entry is added to the set of tasks of the original nodes parent and-node. The worker then releases the lock of the original node, enters the copy of the candidate, releases the lock of the copy and enters the guard handler.

6.3.7 Scheduler

The worker in this state has nothing to do. A new task has to be stolen from another worker. Once a task is found and stolen the worker moves to the associated and-node and the enters the task handler. If all workers are in scheduler state, the whole process stops.

A new task is found by scanning the stacks of the busy workers. If a non-empty stack is found the lock of that worker is taken. Once the lock is taken the worker

Guard instructions	Procedural instructions	Decision instructions
try L G S	call P	switch_on_term Xi, Ls
retry L G	execute P	switch_on_structure Xi Tbl
trust L G	proceed	switch_on_constant Xi Tbl
single L G	allocate S	suspend Xi L S
	deallocate	fail
	guard	

Table 6.1: The guard, procedural and decision instructions

can read the task (if it has not been stolen by someone else) and mark it as stolen. The worker must then add the task to its own stack but must first determine to which and-node the task belongs. This can be done by examining the context stack of the busy worker. Each context entry has a pointer to an and-node and pointers to the task stacks. Before the lock is released the worker must install itself in the right and-node. The right number of context entries are added to the context stack and the stolen task is push on its stack.

The recall and wake stacks are examined first. The continuation stacks are only examined if no wake or recall tasks can be found.

6.4 Instructions

The instructions are divided into four groups: *decision*, *guard*, *procedural* and *term instructions*. Table 6.1 lists the instructions of the first three groups. The abstract machine also makes use of some special instructions to handle abstractions and aggregates. These are discussed in section 6.5 and 6.6.

The instructions are at a first glance similar to the WAM instructions. The decision (switch) instructions and term instructions differ mainly in that they have to handle variables differently. The guard instructions (try, retry, trust) build and access choice-nodes when their WAM counterparts build and access choice-points. The procedural instructions manipulate and-continuations instead of environments and are also responsible for the parallel execution.

I first describe an outline of how a procedure is coded using the guard, procedural and decision instructions. The outline highlights the differences between a WAM coding and a Penny coding. The instructions are then described more in detail.

6.4.1 Encoding

The code of a clause consists of an allocate instruction, a sequence of call instructions, a guard instruction followed by either a sequence of call instructions and a deallocate and execute instruction or a deallocate and proceed instruction.

The code for a definition consist of a sequence of try/retry/trust instructions with code labels referring to the code sequences of the clauses. Below is an outline of the code for a definition.

```
    try L G S
     :

 L allocate S
     :
    call P
     :
    guard
     :
    call P
     :
    deallocate
    execute P
```

The coding scheme is similar to the WAM coding scheme. The only difference is the arguments to the try instruction (guard and size), the arguments to the call instruction (no size) and the unique guard instruction. The function of the arguments are described in the following sections.

If there is only one clause in a procedure the single instruction must be used. The instruction creates the necessary environment for the execution of the guard. This is different from the coding in WAM where the execution can start with the first instruction of the clause.

```
    single L G
 L allocate S
     :
```

The decision instructions (often referred to as switch instructions) are used in *decision code*. This code tries to determine which clauses (or guards) that are worth to explore. If it can be determined that all guards fail, the fail instruction is used.

The decision code can also be used to suspend the execution of a program atom until more information is available. If the decision code can determine that a value of a variable is necessary before a promotion or failure could be possible, the program atom could be suspended on the variable. In this case the suspend instruction is used.

```
L0 switch_on_term 0 L1 L2 L3 ...

L1 suspend 0 L0 S

L2 single L5

L3 fail
     :
```

If it can be determined, possibly by the use of decision code, that a guard is solved, the body can be *directly promoted*. Only the body of the guarded goal is coded resulting in either an allocate instruction, a sequence of call instructions followed by a deallocate and execute instruction or, if there is only one goal in the body, a single execute instruction or, if there are no goals in the body, a proceed instruction. Below is an outline of a definition with a directly promoted body.

```
  switch_on_term  .. L ..
     :
L allocate S
     :
   call P
     :
   deallocate
   execute P
```

Note that an allocate instruction always is present before a guard instruction in the code of a guarded goal and always in the code of a body if there are more than one call/execute instruction in the body. A directly promoted body with only one program atom is coded with a single execute instruction. If the body only contains constraint atoms a single proceed instruction is used.

6.4.2 Guard instructions

The guard instructions are responsible for building the choice-node and the and-nodes in the execution of a program atom. The try and single instructions are executed as the first instructions after a call instruction (if we omit decision code and direct promotion), the retry and trust instructions are executed as the first instruction when the worker enters the instruction handler from the choice handler.

try L G S A choice-node and choice-continuation is created. The choice-node is inserted at the insertion point. The arguments to the goal are stored in the choice-continuation (of size S). The continuation instruction is set to the value of the program counter, i.e. the address of the instruction for the next guarded goal. Finally an and-node is created (with a guard G) and inserted as the first and-node and control is transferred to the code for the first guarded goal (L).

retry L G This instruction is executed when the worker enters the instruction handler form the choice handler, the lock of the choice-node is taken. The argument registers are restored from those saved in the choice-continuation. An and-node is created and inserted as the rightmost and-node in the sequence of and-nodes. The lock of the choice-node is released and control is transferred to the code of the guarded goal (L).

trust L G The trust instruction is equal to the *retry* instruction except that the choice-continuation is changed to a null-continuation.

single L G The instruction is equal to the try instruction but a null-continuation is used as a continuation since there is only one applicable clause.

It is important that the guard operator (G) of an and-node is known as soon as it is created. This allows other workers to enter the and-node and inspect the guard operator even if the guard instruction has not yet been executed.

6.4.3 Procedural instructions

The procedural instructions are responsible for the flow of control in an and-node.

allocate N This instruction creates an and-continuation with room for N permanent registers. The instruction be used even for a clause with no permanent variables since the continuation also includes the continuation code pointer. The and-continuation is inserted at the insertion point and becomes the current and-continuation.

call P A call instruction initiates the call of a program atom. The continuation program pointer of the current and-continuation is updated. If the insertion cell to the left of the current insertion cell is dead it becomes the current insertion cell otherwise a new insertion cell is created to the left of the current cell. The new insertion cell is marked as free. A continuation task is added for the and-continuation and the worker starts decoding the instructions from the definition.

deallocate The current and-continuation is discarded and the insertion cell is marked as free. The next instruction is either an execute or proceed instruction.

execute P The instruction is different from the call instruction only in that it pertains to the last goal in the body of a clause. Therefore the continuation is not saved nor is a new insertion cell created. The insertion cell comes either from a discarded and-continuation or is the first insertion point created in the and-node.

proceed The last instruction of a body. The insertion cell is marked as dead and the execution continues in the task handler.

guard The guard instruction is the last instruction of the guard. The current insertion cell is marked as dead and execution continues in the task handler after having updated the program counter of the current and-continuation.

6.4.4 Decision instructions

Decision code is probably the most important optimization of the abstract machine. In the Penny system only limited decision code has been implemented since a more carefully study and design has been worked out by Per Brand [11].

switch N Ls The switch instruction examines the term in argument register N and then proceed execution at one of the labels. The switch instructions can of course come in any flavor, the most common is the switch-on-term instruction that examines the type of the term, other instructions can examine the value of a specific term etc.

The description of the abstract machine is not depending on the switch instructions. Any sequence of switch instructions must eventually end in a sequence of try/retry/trust instructions, a single instruction, a suspend instruction or a fail instruction.

suspend Xi L S A choice-node and choice-continuation is created. The choice-node is inserted at the insertion cell. The arguments to the goal are stored in the choice-continuation of size S. The continuation instruction is set to the address L for the decision or guard code that should be executed when the goal is woken. A recall suspension is added to the variable in argument register Xi and execution continues in the task handler.

fail The fail instruction ends a sequence of decision instructions if it can be determined that all guards fail. Execution continues in the fail handler.

In general one would like to suspend a goal on a set of variables or even better on all variables in one of many sets. The simple suspension mechanism in the Penny system is easily extended but this has not been done. The problem is not in the abstract machine but in the compiler. It's a major effort to extract the right sets of variables in an efficient way.

6.4.5 Term instructions

Get instructions	Put instructions	Unify instructions
get_variable xi/yi xi	put_variable xi/yi xi	unify_variable xi/yi
get_value xi/yi xi	put_value xi/yi xi	unify_value xi/yi
get_constant C xi	put_constant C xi	unify_constant xi/yi
get_structure S xi	put_structure S xi	unify_structure xi/yi
get_list xi	put_list xi	unify_list xi/yi
get_nil xi	put_nil xi	unify_nil xi/yi
	put_void xi	unify_void xi/yi

Table 6.2: The term instructions

Table 6.2 lists the term instructions. The term instructions are used to construct terms and to do open-coded unification. The term instructions are unification specific i.e. specialized for equality over rational trees (get_variable, put_value and put_variable are general).

Unify instructions can be executed in either *read mode* or *write mode* depending on if a new structure is created or if an existing structure is inspected. The structure pointer and write mode flag are only valid during a uninterrupted sequence of term instructions and need not be saved if the worker leaves the instruction handler or executes a procedural or guard instruction.

The arguments of the term instructions refer to the argument registers of the worker (Xi) and the permanent registers (Yi) of the current and-continuation.

In the unification process, new wake and recall tasks might be added to the the task stacks. This happens if a local variable with suspensions is bound or if an external variable with a local suspension entry is bound. In the latter case the suspension entry is transformed into a unification entry. This means that suspensions, on the level above, that were referring to the suspension entry after the conditional binding is referring to a unification entry. If unification fails the worker proceeds to the fail handler. The next chapter discusses in detail how the binding scheme works.

6.4.6 Get instructions

get_variable Xi/Yi, Xj The value in register Xj is moved to register Xi/Yi.

get_value Xi/Yi, Xj The value in register Xj is unified with the value in register Xi/Yi.

get_constant C, Xj The constant value C is unified with the value in register Xj.

get_structure F, Xj If register Xj contains a variable, then a template for a structure with functor F is created, and it is bound to the variable. Also, the structure pointer is set to refer to the first argument of the structure, and the execution enters write mode. If the value in Xj is a structure, and its functor is equal to F, the structure pointer is set to refer to the first argument of the structure. Otherwise, failure is performed.

get_list Xj If register Xj contains a variable, then a template for a list cell is created, and it is bound to the variable. Also, the structure pointer is set to refer to its first argument, and instruction decoding enters write mode. If the value in Xj is a list, the structure pointer is set to refer to the first argument of the list. Otherwise, failure is performed.

get_nil Xj The constant value nil is unified with the value in register Xj.

6.4.7 Unify instructions

The unify instructions are executed in read or write mode.

unify_variable Xi/Yi In read mode, the value referred to by the structure pointer is moved to register Xi/Yi. In write mode, a new variable is created and is stored both in register Xi/Yi and in the argument referred to by the structure pointer. In both cases the structure register is incremented.

unify_value Xi/Yi In read mode, the value referred to by the structure pointer is unified with the value in register Xi/Yi. In write mode, the value in Xi/Yi is stored in the argument referred to by the structure register. In both cases the structure pointer is incremented.

unify_constant C In read mode the value referred to by the structure register is unified with the constant C. In write mode the constant C is stored in the argument referred to by the structure pointer. In both cases the structure pointer is incremented.

unify_structure F In read mode the value referred to by the structure register is unified with a template for a structure with functor F. In write mode such a template is created and stored in the argument referred to by the structure pointer. In both cases the structure pointer is set to refer to the first argument of the structure.

unify_list In read mode the value referred to by the structure register is unified with a template for a list structure. In write mode such a template is created and stored in the argument referred to by the structure pointer. In both cases the structure pointer is set to refer to the first argument of the list.

unify_nil In read mode the value referred to by the structure register is unified with the constant nil. In write mode the constant nil is stored in the argument referred to by the structure pointer. In both cases the structure pointer is incremented.

put_abstraction Definition, Xi
call_abstraction Abstraction Arguments
execute_abstraction Abstraction Arguments

Table 6.3: Instructions to handle abstractions

unify_void In read mode the structure register is incremented. In write mode, a new variable is created and stored in the argument referred to by the structure pointer. The structure pointer is then incremented.

6.4.8 Put instructions

put_variable Xi/Yi, Xj A new variable is created and is stored both in register Xi/Yi and in register Xj.

put_value Xi/Yi, Xj The value in register Xi/Yi is stored in register Xj.

put_constant C, Xi The constant C is stored in register Xi.

put_structure S, Xi A template for a structure with functor S is created and placed in register Xi. The structure register is set to refer to the first argument of the structure, and the worker enters write mode.

put_list Xi A template for a list cell is created and placed in register Xi. The structure register is set to refer to its first argument, and the worker enters write mode.

put_nil Xi The constant nil is stored in register Xi.

put_void Xi A new variable is created and is stored in register Xi.

6.5 Abstractions

It is easy to handle procedures as data. We need an instruction to create an *abstraction* and an instruction to *apply* the abstraction to some arguments.

An abstraction is represented by a structure that holds a code pointer, an arity and a tuple of m free arguments, it is applied to a set of n formal arguments. The free arguments are given when the abstraction is created the additional formal arguments are given when the abstraction is applied.

put_abstraction D, Xi The instruction allocates a template for an abstraction with definition D and places it in register Xi. The definition holds the number of arguments to the abstraction. The structure register is set to refer to the first argument of the abstraction, and the worker enters write mode.

A body containing an abstraction

```
  :
X = (A,B)/foo(A, C, B, D),
  :
```

is constructed with the sequence

```
  :
put_abstraction Lx Xi
unify_value Xc
unify_value Xd
  :
```

where the definition of the compiled auxiliary procedure `aux/4` is located at Lx and the values of C and D is found at Xc,and Xd. The auxiliary procedure

```
aux(A, B, C, D) :- -> foo(A, C, B, D).
```

takes care of the ordering of the arguments. The formal arguments (A, B) are placed first followed by the free arguments (C, D). The definition of the procedure holds apart from the arity the number of formal arguments needed.

call_abstraction Xi Xj The instruction takes an abstraction in register Xi and a list of arguments in register Xj. The number of elements in the list plus the number of free arguments of the abstraction should be equal to the arity of the definition of the abstraction.

The application of an abstraction

```
  :
apply(Abstr, Arg),
  :
```

is compiled using the call_instruction

```
  :
call_abstraction Xi Xj
  :
```

The instruction places the elements of the list in the argument registers 0 to n and place the arguments stored in the abstraction in argument register $n+1$ to $n+m$. The continuation program pointer of the current and-continuation is updated and a new insertion cell is, if necessary, created to the left of the current and-continuation. A continuation task is added for the and-continuation and the worker starts decoding the instructions from the definition.

If the list of arguments is given at compile-time for example as follows:

```
   :
apply(X, [a,b]),
   :
```

the call can be compiled to

```
   :
put_constant a 0,
put_constant b 1,
call_abstraction Xi 2
   :
```

The arguments (a,b) are put in the argument registers 0 to 1. The second argument to the call_abstraction instruction is needed in order for the instruction to know where the free arguments should be placed and to verify that the total number of arguments matches the arity of the definition.

If the application of the abstraction is the last call in the body the execute_abstraction instruction is used. This is similar to the relation between the call and the execute instructions. This means that we can write tail recursive procedures where the last call is an application of an abstraction.

The apply instruction needs the suspension mechanism described in section 6.4.4. If the abstraction or list of arguments are not known a suspensions has to be created. The suspension is represented with a choice-node with two arguments, the abstraction and the list of arguments, the instruction pointer of the choice-continuation points to a single execute_abstraction instruction that is available to the emulator.

6.6 Aggregates

The coding of aggregates make use of the three addition instructions listed in table 6.6.

The abstract machine also make use of a unique data structure called a *collector object*. The object serves as a representation of the port structure created by the aggregate rule. The collector object holds a pointer to the term that is referred to

collector Xi, Xj
collect Yi, Yi, Yk
unit Yi, Yj

Table 6.4: Instructions to handle aggregates

as the *bag*. Solutions can be added to the bag by destructively updating the object. When a collect or unit guard is executed the the collector object is located using the collect and unit instructions.

collector Xi, Xj The instruction creates an *collector object* in argument register Xi and lets it refer to the term found in Xj.

collect Yi, Yj, Yk The instruction locates a collector object in Yi, places the bag in Yj and lets the collector point to a new variable stored in Yk.

unit Yi, Yj The instruction locates a collector object in Yi and places the bag in Yj.

The instructions makes it possible to implement different forms of aggregates. To implement the **bagof/2** procedure the bag term is unified with a cons cell and that the collector object is updated to point to the cdr of the cons cell. The code for the **bagof/2** procedure is listed below.

```
procedure  'bagof'/2           % x0 holds the abstraction
                               % x1 the bag

        collector x1 x1        % create a collector
                               %   let the collector point to the bag
                               %   store the collector in x1

        try L0 order 2         % try the collect guard
        trust L1 unit          % trust the unit guard

L0      allocate 3             % allocate a continuation
        get_variable y2 x1     % save the collector in y2
        get_variable x2 x0     % move the abstraction to x2
        put_variable y1 x0     % create a variable in y1 and x0
        call_abstraction x2 1  % apply the abstraction to x0

     guard                     % when an  answer is found
        collect y2 y2 y0       % find the collector in y2
                               %   place the bag in y2
                               %   create a new tail variable y0
```

```
        put_value y2 x2        % move the term to x2
        get_list x2            % unify it with a list
        unify_value y1         %     car is answer
        unify_value y0         %     cdr is the tail
        deallocate             % over
        proceed                %     and out


L1      allocate 1
        get_variable y0 x1     % save the collector in y0

     guard                     % all solutions have been found
        unit y0 y0             % find the collector in y0
                               %     place the bag in y0
        put_value y0 x2        % place the bag in x2
        get_nil x2             % unify it with nil
        deallocate             % over
        proceed                %     and out
```

Chapter 7

The Binding Scheme

THIS CHAPTER DESCRIBES the binding scheme. The binding scheme is the most efficient bindings scheme yet published for a concurrent constraint programming language. A self contained version of the chapter has been published in New Generation Computing November 1995.

7.1 Multiple environments

The binding environments should be represented in a way that induces as little overhead as possible but at the same time allows several processes to execute in parallel without too much interference. The perfect solution does of course not exist. Any solution make some sacrifices; the question is when the penalty should be taken.

Several solutions to the problem of maintaining multiple binding environments have been published ([25] gives a good overview and comparison). These solutions are however designed for or-parallel Prolog systems, where different bindings exist in different or-branches in an execution state.

In the Penny implementation, or-nodes are not used; alternative environments are created by explicitly copying all structures involved. The implementation need therefore only handle bindings on different levels in the configuration. Since the levels in an AKL configuration normally are fewer than the levels of or-nodes in a Prolog execution, a simple scheme can be used. The scheme is close to the computation model and places little overhead on the most frequent operations.

7.2 Term representation

Terms are represented by tagged pointers. In this chapter it is sufficient to distinguish unconstrained variables (UVA), constrained variables (CVA), term references (REF), atoms (ATM) and structures (STR).

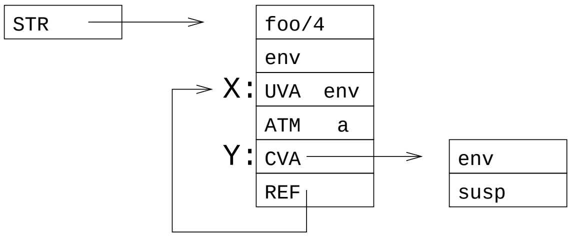

Figure 7.1: The representation of a term

An unconstrained variable holds, apart from its tag, a reference (env) to the home and-node. The and-node reference, points to the environment identifier of the and-node. A constrained variable holds a reference to a structure. The structure holds an environment reference and a list of *suspensions* (susp). All variables are created as unconstrained variables; only if a suspension is added to the variable, does it turn into a constrained variable.

Atoms hold only an identifier, typically a short integer or an index to an atom table. Structures hold, apart from the functor and its arguments, an environment reference. The term reference is used for variable-variable bindings. The dereferencing procedure follows any term references until a structure, an atom or a variable is found. Figure 7.1 shows an example of how the structure "foo(X,a,Y,X)" (where the variable "Y" is constrained) is represented.

Note, that the representation of variables is different from the representation of variables in WAM [72] where unbound variables are represented by a self-reference. Care must be taken when a variable is bound to another variable or copied to a register. In WAM the value of any term is simply copied to a register. If the term is a variable the copied value automatically becomes a reference to the variable. This is of course not possible if a variable has its own tag and an environment reference. Instead, a new term reference has to be created. This implies that a worker must first determine the type of term, before the copy operation is performed. The self-reference is used, as described in Section 7.5, as a lock.

7.2.1 Who is local?

Each variable belongs to an and-nod, referred to as the home of the variable. When a variable is created it is given an environment reference that refers to the current and-node. It is thus easy to check whether a variable is local to an and-node. A problem does arise when an and-node is promoted and all variables should become local to the parent and-node. There is no efficient way of finding all variables that are local to the and-node and update their environment references, instead a forward reference is added to the promoted and-node.

To check whether a variable is local, the environment reference of the variable is compared with the environment identifier of the current and-node. If they are equal the term is local, otherwise the and-node is examined. If it is a promoted and-node the forward pointer is followed until an un-promoted and-node is reached. The environment identifier is then compared to this and-node. Environment references can be updated any time during execution, there is no need to keep an reference that refers to a promoted and-node.

This scheme is similar to the scheme designed in a sequential implementation of Concurrent Prolog [46] where variables also have to be identified with a level in the computation.

7.3 Binding lists

Bindings to local variables can never be removed and can therefore be recorded in place, i.e. the value of a variable is permanently replaced by the binding. Local bindings to external variables must only be visible in and below the and-node in which the binding occurs and are therefore recorded in a *binding list* local to that and-node. If an and-node has a local binding on an external variable X it means that the and-node is suspended on the variable X.

A binding list consists of an entry for each external variable that is locally bound in or below the and-node. There is at most one entry for any variable in a list, but there can of course be many entries for a variable in different lists. An entry contains, a pointer to a hanger, a reference to the constrained variable, and either a term or a list of suspensions. The hanger holds, as described in the previous chapter, a reference to the environment identifier of the and-node. An entry that contains a list of suspensions is called a *suspension entry*. An entry that contains a term is called a *unifier entry*, the term is the local binding of the variable.

A suspension contains a reference to an entry. A suspension, on a constrained variable, refers to an entry in an and-node immediately below the home of the variable. A suspension, of a suspension entry, refers to an entry of an and-node immediately below the and-node to which the suspension entry belongs. The hierarchical structure of the suspensions, is used both to maintain locality of suspensions and to detect stability. Figure 7.2 shows a constrained variable “ Y” and the hierarchy of suspensions. The hangers are not shown since these are not important for the binding scheme.

7.3.1 Lookup

To lookup the current value of a term, normal dereferencing must first be done i.e. follow any term-reference pointers until a non-reference value is found. If the found value is a variable, the environment reference is examined. If it is a local variable, the variable is unbound. If it is external, it could be unbound, but it could also be locally bound.

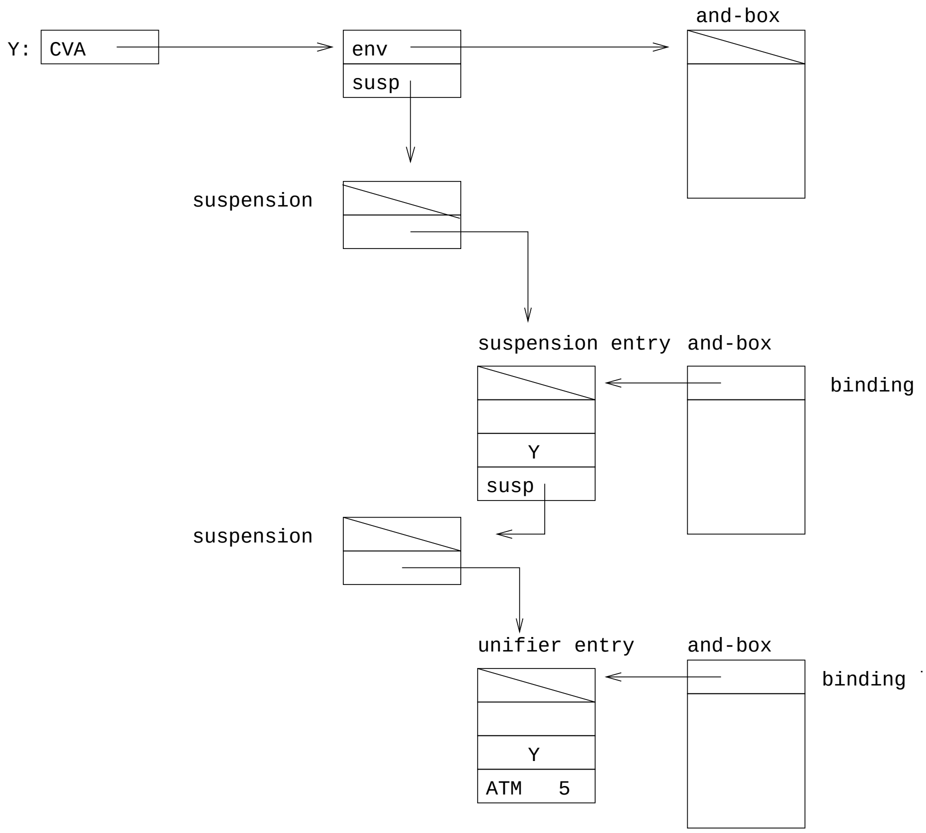

Figure 7.2: The hierarchy of suspensions

In order to find the local binding of an external variable, each binding list on the path from the current and-node to (but not including) the home of the variable, must be examined. The binding lists are examined in order, starting with the binding list of the current and-node and moving upwards. There might exist several entries for the variable, albeit only one in each binding list. The first entry that is found, if any, is the only entry that is interesting, the rest can be ignored.

If no entry is found, or if the found entry is a suspension entry, the variable is unbound in the current and-node. If a unifier entry is found, the entry holds the local value of the variable. Note, that the local value might be a reference term or a variable, in that case the dereferencing and lookup procedures has to be performed again. An important property of the binding scheme is, that no circular bindings are created neither by term reference pointers, nor through binding list entries. The following sections will describe, how circular bindings are avoided.

Note that there might be conflicting bindings on the path from the current and-node to the home of the variable. A suspension entry in one binding list can also shadow a unifier entry in another binding list. How this can occur and why it is not a problem will also be described in the following sections.

7.3.2 Stability

The binding scheme defines an efficient way of detecting, whether any computation in a subtree, formed by an and-node, is suspended on a variable that is external to the and-node. It does not define how to keep track of deterministic executions in the configuration, this is described by the abstract machine.

To understand how stability is detected, the properties of the binding scheme must be clear. The most important, is how the hierarchy of suspensions will reveal the unifier entries in a subtree. For each unifier entry of a variable, there is at least one suspension entry, in every binding list, on the path from (but not including) the and-node of the unifier entry, to (but not including) the and-node that is the home of the variable.

A *living entry* is, either a unifier entry of a living and-node, or a suspension entry that has a suspension, that refers to a living entry. Suspensions, can also refer to entries that belong to promoted, failed or pruned and-nodes.

Stability of an and-node can be detected by examining the entries in the binding list.

An and-node is unstable if and only if, it holds a living entry.

Note, that the stability check is limited by the suspension entries of the current and-node. There is no need, to traverse the subtree looking for unifier entries for external variables, since any such entry would reveal itself, as a suspension entry in the and-node. The price for this is paid, when local bindings to external variables are added but, as will be shown in the Section 7.6, the number of local bindings is small.

7.3.3 Adding a binding

A local variable is bound to a structure or atom by replacing the value of the variable with the atom or structure. A local variable is bound to another variable by replacing its value with a term reference to the other variable. If the bound variable is constrained, the suspended and-nodes are *woken* by turning the suspensions into wake tasks.

An external variable, with no local binding, is bound to a structure or atom by adding a unifier entry to the local binding list. If the binding list already contains a suspension entry for the variable, this entry must be reused and, the suspensions of the entry are turned into wake tasks.

Figure 7.3 shows the hierarchy after a worker positioned in the middle and-node has added a local binding $Y = 4$. Notice, that we now have conflicting information in the path, from the lowest and-node to the home of Y. A worker that is positioned in the lowest and-node would find the local binding of $Y = 5$ and be unaware of the

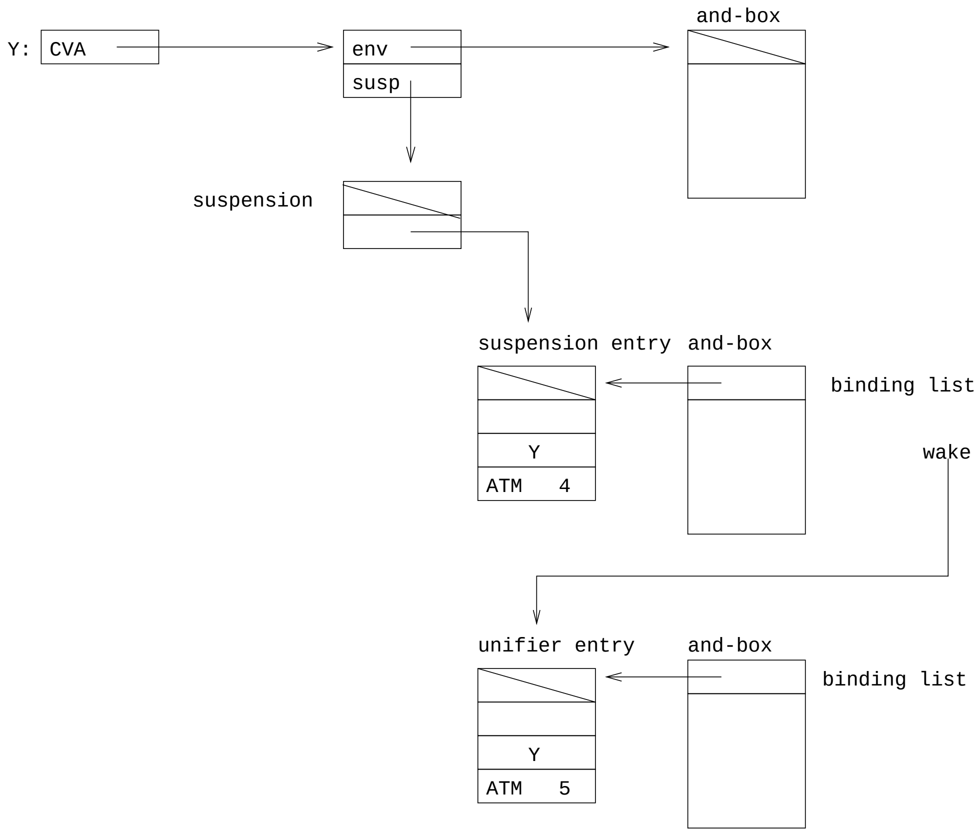

Figure 7.3: Adding a local binding to an external variable

conflict. It is up to the worker, that added the binding, to wake the lowest and-node and fail the computation.

Figure 7.4 shows the situation after a worker, in the home and-node of Y, has added a binding $Y = 42$. It is, of course, up to this worker to wake the middle and-node and fail the computation. An odd, but harmless, situation has occurred where the worker in the lowest and-node now has the view that Y is equal to 42, even though there exist a local contradictory binding for Y. This situation is harmless, since any execution within the lowest and-node will not alter the state outside the and-node. The and-node could of course be promoted, if it is the only remaining and-node below a choice-node. In this case the middle and-node will fail, since the contradictory constraint is promoted, but the result will be the same, if the last and-node is removed from the choice-node.

If no suspension entry is found, in the current and-node, the hierarchical structure of suspensions must be updated. Suspensions are added to each binding list, from the parent and-node to (but not including) the home of the variable, and to the variable itself. A suspension is added to a binding list, by adding it to an already available suspension entry for the variable, or by creating a new suspension entry. If the variable is an unconstrained variable, it is turned into a constrained variable. Notice, that the process of updating suspensions entries, only has to consider the

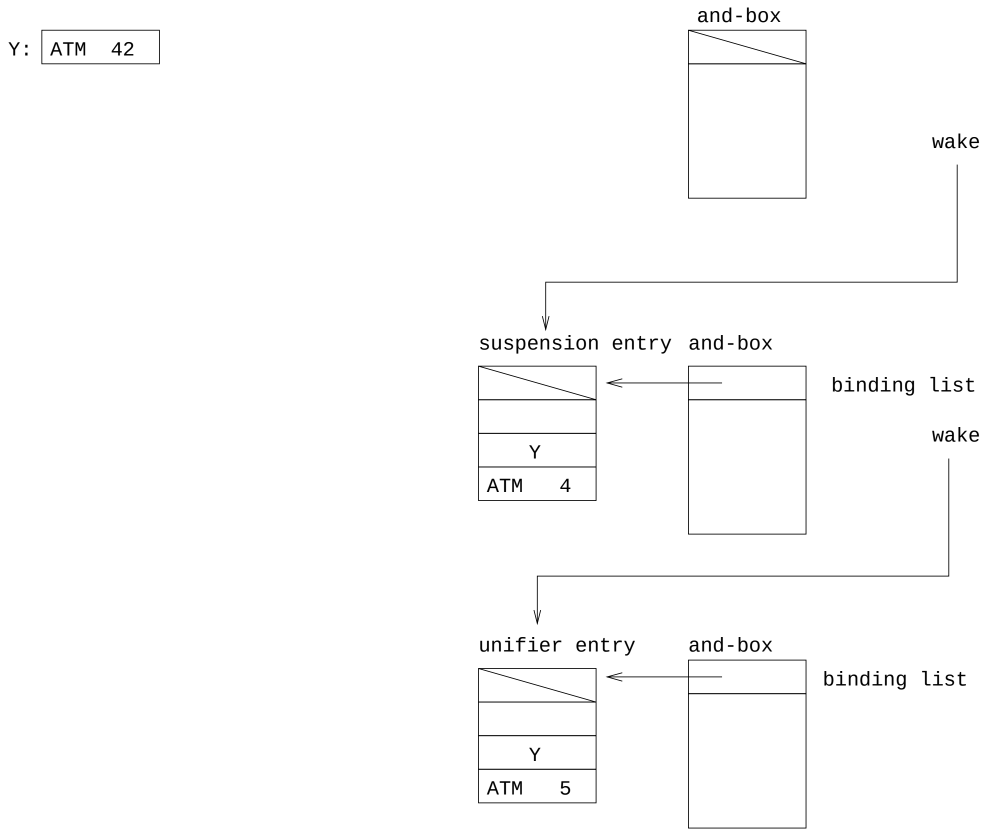

Figure 7.4: Adding a binding to a local variable

binding lists, in the path from the current and-node to the first and-node that contains a suspension entry for the variable.

7.3.4 Variable-variable

In a variable-variable binding where both variables are external, care must be taken. The *least external* (the variable whose home and-node is closer to the current and-node) must be bound to the *most external*. If the most external is bound to the least external, the home and-node, of the least external variable, will contain a suspension entry for the most external variable. This would mean, that the and-node of the least external variable would be treated as unstable, whereas, in fact, it is not.

If both variables belong to the same and-node, either direction can be chosen, but suspensions must be added for both variables. This is necessary in order to be able to wake the and-node, regardless of which variable that is bound. An alternative approach would be, to always wake the suspended and-nodes of both variables in a variable-variable binding. The benefit of such an approach would be, that there always would be at most one suspension referring to an entry. The overhead of adding the two suspensions cannot be neglected since external variable-variable bindings are rare. The deficiency would be that more wake tasks would be created, even when

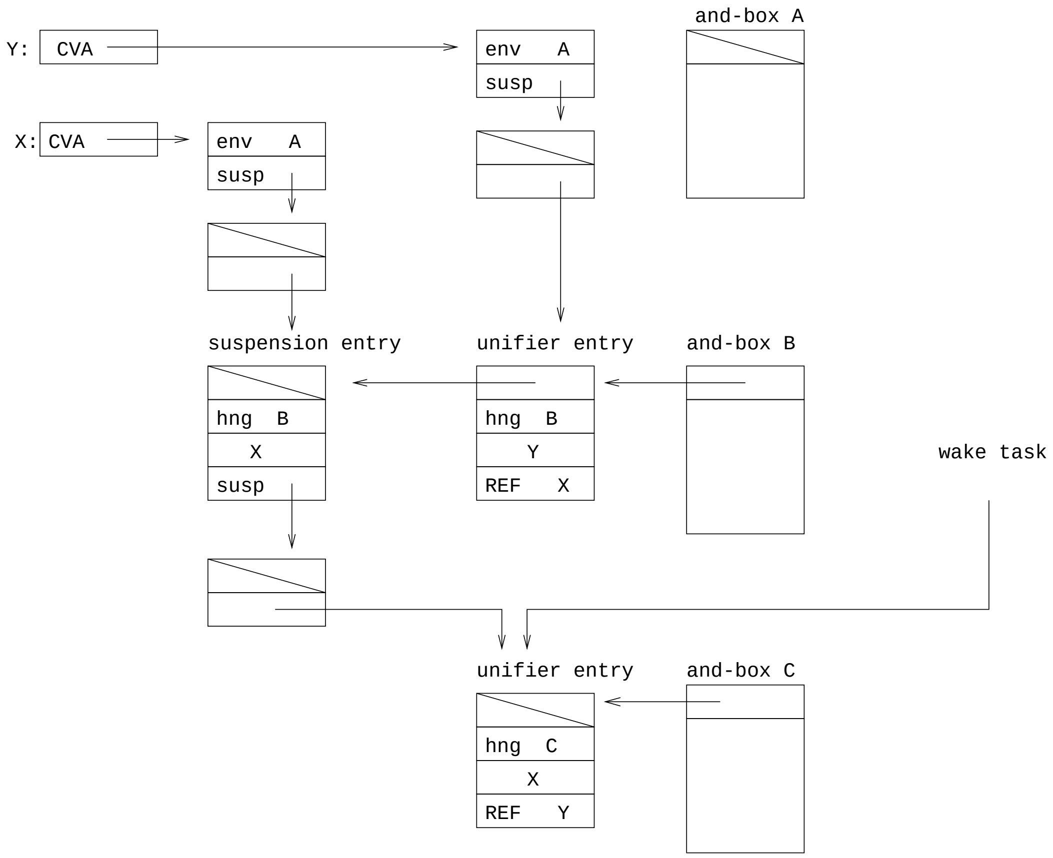

Figure 7.5: A circular binding

two local variables were bound to each other. This could be, however, a cheap price to pay if the management of entries could be simplified.

Note, that circular bindings on external variables, temporarily, might exist. Consider the three levels of and-nodes, with two variables X and Y that belong to the uppermost and-node A, as shown in Figure 7.5. If a local binding $X = Y$ is added to the lowest and-node C the result would be a unifier entry, in the and-node C, and two suspension entries (one for each variable), in the middle and-node B. If another worker then adds a local binding $Y = X$, to the middle and-node B, the suspension entry of Y is turned into a unifier entry.

The result is that a circular binding has been created. A worker that tries to find the value of X, in the lowest and-node C, will now be stuck in a loop. The worker that added the binding, in the middle and-node B, is responsible for waking the lowest and-node. When this is done, the entry in the lowest and-node will be removed, and the correct value of X (unbound) can be found.

7.3.5 Waking an and-node

When an and-node is woken the worker knows which entry is to be re-examined and this entry is removed from the binding list. If the entry is a unifier entry the unification is retried. If the entry is a suspension entry the suspended and-nodes are woken.

Note that the hierarchical structure of suspensions gives perfect information about which and-nodes that must be woken. It is not necessary to examine an and-node to determine if it is below the current and-node and thus is to be woken.

7.4 Unification

Unification of two terms would be simple, if it was not for the fact that some terms might be circular. The traditional Prolog solution to this this problem, is to destructively change one of the terms to refer to the other term. The changes are kept on a trail, so that the original term can be restored.

The Prolog approach cannot be used in the Penny system, since unification of structures could be local to a guard and should not be visible outside the guard. Instead of changing the representation of one of the terms, a table is used to record equalities of structures. Each time two structures are unified, the (hash) table is consulted to see if these two terms have already been unified. The entries in the table are only kept during one invocation of the unification procedure. No information is kept in the and-node for subsequent invocations.

An interesting optimization can be done if each structure, as described, also has an environment reference. If a local structure is unified with another structure (local or external), the local structure can be changed to refer to the other structure. These changes need not to be trailed nor reset since the unification is unconditional for the structure.

The same technique could actually be applied, both in flat committed choice systems, such as KL1, and in Prolog systems. In a flat committed choice system all structures belong to the same level. This means that unifications can be done destructively. In a Prolog system the structures, that have been created after the last choice-point, can be destructively changed, since the whole structure will be reclaimed upon backtracking.

The optimization is not so important since unification of larger structures is rare. The access to a hash table could, however, be costly, if they are infrequent. The table, most certainly, will not be in the cache.

Andreas Podelski and Peter van Roy have presented [52] an algorithm called “the beauty and beast” for tests of entailment and dis-entailment. It is an algorithm that avoids the $O(n^2)$ complexity (in the number of constraints), that is inherent in the presented scheme.

7.5 Locking

The description, above, does not consider how the binding scheme works in a parallel implementation, where more than one worker can access and modify the representation. To guarantee a consistent state, a locking scheme is needed. This section describes a locking scheme, that allows several workers to add new bindings in parallel.

7.5.1 Self reference

A variable is locked by, atomically, exchanging its current value with a reference to the variable itself. If a self-reference is returned, the locking operation is retried. The circular reference will lock, the access to the variable. This scheme has the interesting property, that any worker that tries to dereference the value, will be stuck in a loop until the lock is released. The locking scheme, does not change the dereference procedure.

This locking scheme has been proposed, by Jim Crammond, as a technique for locking multiple variables in a FCP [16]. The technique has also been used in the Super Monaco system, implemented by Evan Tick and his group [37].

7.5.2 Hazards

There are two possible hazards in the binding scheme: circular references and deadlock situations. If two local variables are unified, the binding scheme does not specify the direction of the binding. Two workers might create a circular reference (one worker places a reference in X to the variable Y, while another worker places a reference in Y to X). If two external variables, of the same and-node, are unified, suspensions should be added to both variables. Both variables must therefore be locked, and a deadlock situation might occur.

To make the binding scheme deadlock free, and to avoid circular bindings, a well defined order of variables must exist. The order can be defined by the address of the variable. The order need only be preserved during the lock operation, not during the whole execution, so a garbage collection, that changes the order between variables, does not cause a problem. If the order of variables is considered whenever more than one variable is locked, the scheme will be dead-lock free.

7.5.3 Adding an entry

Note, that by locking the variable itself, no other worker can add an entry for the variable during the binding operation. This prevents two workers from, simultaneously adding a local binding for the same variable, even if the bindings are in different parts of the configuration. This is a drawback, but can also be used, as will be shown, as an advantage.

Consider, binding a variable Y to a constant. First the variable is locked. By locking the variable the worker has exclusive right to the variable, so no other binding operations can be performed. Then the type and locality of the variable is examined, if it is a local variable the value of the constant is written to the position of the variable, and suspensions are turned into wake task.

If the variable is external, the binding lists are examined, in order to locate any local bindings of the variable. The lookup procedure, as said before, only has to consider the first found entry. If a local binding is found, the lock of the variable is released, by writing back the original value of the variable. The binding procedure then starts all over, trying to unify the local value with the constant.

If no local binding is found, the binding list of the current and-node is examined. If the list contains a suspension entry for the variable, it is turned into a unifier entry and the lock of the variable is released. If no entry is found a unifier entry is added to the list and the suspensions in the path form the current and-node to the home of the variable are updated. The lock of the variable can then be released.

Note that no other worker is allowed to add or change the entries for the locked variable so this can be done with a minimum of locking. A lock is of course needed to synchronize the manipulation of the binding lists but this only has to be taken when a new entry is added or removed. There is no need to take the lock of a list in order to look for entries of a variable.

7.6 Evaluation

The main advantage of the described binding scheme is that a worker can move freely in the configuration since it does not need to update any private information. This allows for fast task switching. The explicit representation of the constraint stores also makes bindings immediately visible to all workers; all workers have a consistent view of the bindings in the configuration.

A disadvantage is the non-constant time operations: to access or to add a new binding can in the worst case be a costly operation. In practice it does not cause any problem. The majority of variable accesses are made to local variables, in which case the binding is found in place, or to variables that are local to the parent and-node, in which case only one binding list is examined.

To evaluate the binding scheme statistics were gathered from five programs. The programs were executed in a prototype implementation, running with one worker. The programs are:

Program	local	one level	two levels
compiler	97%	3%	0%
waves	65%	35%	0%
life	77%	23%	0%
scanner	87%	12%	1%
knights	81%	18%	1%

Table 7.1: The frequency of local and external bindings

compiler The compiler compiling itself (about 3000 lines of AKL code). The program uses concurrency and deep guards. The deep guards are used for convenience, the program could be rewritten with only flat guards.

waves Waves in a torus (5 generations, dimension 9), a flat program (originally written in Strand by Ian Foster, translated to KL1 by Evan Tick).

life The game of life (10 generations in a 40 times 40 toroid), a flat program written in AKL. Each cell is implemented as a process that depends on all its eight neighbors.

scanner Find a pattern in a grid given x-ray information from rows, columns and diagonals. A program that uses both concurrency and non-determinism.

knights The knights tour (first solution on an 8 times 8 board), a program that uses both concurrency and non-determinism.

The programs can be divided into three categories. The **compiler** is a real-life program i.e. not only written for benchmark purposes. The **waves** and **life** programs are both flat committed choice programs, and are interesting since neither can be executed depth-first without any suspensions. Almost all of the traditional benchmarks for flat committed choice languages can be executed depth-first without any suspensions. Such programs,, although they represent a big group, were not included since a single worker execution will not use any deep bindings. The **scanner** and **knights** are programs that utilize both concurrency and non-determinism, AKL's crowning glory.

Table 7.1 shows the percentage of bindings made to local and external variables. As is clearly seen, the majority of bindings are made to local variables. Local bindings of external variables, are almost always made to variables that belong to the parent and-node. Bindings that span two levels are rare.

Once a local binding has been made it can of course be accessed many times. If it is accessed in the same and-node in which the binding is made, only one binding list is searched. Only if the access is even further down the tree, need more than one binding list be traversed. Table 7.2 shows how many levels that have to be searched

program	one level	two levels
compiler	100%	-
waves	100%	-
life	100%	-
scanner	100%	0%
knights	94%	6%

Table 7.2: The number of levels searched

program	0	1	2	3	4	5
compiler	90%	5%	5%	-	-	-
waves	94%	6%	-	-	-	-
life	42%	47%	14%	13%	3%	$< 1\%$
scanner	98%	$< 1\%$	$< 1\%$	$< 1\%$	$< 1\%$	$< 1\%$
knights	42%	33%	23%	1%	$< 1\%$	$< 1\%$

Table 7.3: The number of elements traversed in each binding list

before a binding is found or the home and-node is reached. As is clearly seen, in almost all accesses to external variables only one binding list is traversed.

One may also wonder if a list of bindings is efficient enough and if it would not be more efficient to use a hash table. In Table 7.6 the number of elements traversed in each binding list for each variable access is shown. The number of elements is as one can see small and except for the life benchmark a maximum of two elements have to be traversed in 98% of the cases. The need for a more efficient representation is hardly imminent.

The use of the environment reference to determine if a variable is local or external is another possible hazard. To dereference the environment is a non-constant time operation but it turns out that this is an infrequent operation. In the tests described above only two thousand, out of two and a half million ($< 0.1\%$), environments had to be dereferenced, in all other cases the environment references pointed directly to non-promoted and-nodes.

External variable-variable bindings are as described complex. It turns out that these are used so infrequently that one has to construct artificial programs in order to test the more complex cases. There is no need to put to much work into an efficient implementation of these cases. It is more important to keep the implementation as simple as possible in order to be able to implement them correctly.

7.7 Related work

Gopal Gupta and Bharat Jayarama [25] describe criteria for or-parallel execution models of logic programs. Comparing the Penny scheme with or-parallel Prolog models is however not straightforward. The implementation of choice splitting sacrifices constant-time task creation whereas the binding scheme sacrifices constant-time variable access. The criterion in the development of the scheme has been that local variable access should be a constant-time operation. In order to achieve this, constant-time choice splitting or constant-time task switching had to be sacrificed. A copying strategy for choice splitting was chosen since it simplifies the binding scheme.

Binding schemes for Concurrent Prolog have, as in AKL, to deal with multiple levels of bindings and suspension of goals. The "deep scheme" proposed by Sato, H. et al. [56] is similar to the Penny binding scheme. That scheme also uses a hierarchy of binding lists that have to be searched for each access of an external variable.

The ParAKL [48] implementation of AKL uses a binding scheme based on the PEP-Sys hashing scheme. A hierarchical structure of hash tables is used to represent the constraint stores. The hierarchical structure reflects how work has been spawned and not the structure of the configuration. If two workers are executing in the same and-node the last spawned will record its bindings in a private hash window. Research has been done on compile time analysis for stability detection [47].

The sequential prototype implementation of AKL [34] uses a trailing scheme to implement the constraint stores. All bindings are made in place but external bindings are trailed. The implementation uses only a dirty bit to detect stable and-nodes with the consequence that once an and-node has become unstable it will be marked as unstable even if it later becomes stable. A similar scheme is used in the DFKI-OZ system but the stability check has been limited to and-nodes immediately below solve combinators.

Andreas Podelski and Gert Smolka has presented situated simplification [51] that is a formal description of a system that is similar to the Penny binding scheme. The description differs mainly in that the local binding of a variable is accessed through the variable using the environment as an index. This is of course an implementation decision but to determine stability the local bindings must be accessible given the and-node and not vice versa.

Chapter 8

Implementation

IN THIS CHAPTER I describe the implementation specific details of the Penny system. The chapters includes descriptions of locks, tags, generic objects, memory management, garbage collection, copying procedures etc.

8.1 The Target Architecture

The abstract machine is designed to be implemented on a shared-memory multiprocessor. The design assumes that the execution state can be shared among workers, and that all workers have uniform access to all parts of the representation.

The implementation is optimized for the actual hardware that we have. The hardware is however a good representative for modern shared-memory architectures. The optimizations should be applicable to similar machines even if some parameters are machine specific.

8.1.1 The Machine

The SPARCCenter 2000 (SC2000) is a bus-based shared-memory multiprocessor from Sun Microsystems. Each processor has two on-chip caches, one for instructions and one for data. These are generally small, because on-chip area is a scarce resource. On the SC2000's processors, 50MHz SuperSPARCs, the data cache is 16Kbytes, four-way associative with 32 byte cache lines. The instruction cache is 20Kbytes, five-way associative with 64 byte cache lines. Despite their small size, the first level caches consume half of the SuperSPARC's three million transistors.

The processors connect to an off-chip cache, the second level cache, which on the SC2000 is 2Mbytes, direct-mapped with 64 byte cache lines. These caches are connected to a bus, whereby they can communicate with the main memory and/or other caches. This communication is controlled by SuperCache controllers and "Bus Watcher" chips. There are actually 2 buses on the SC2000, dual 40MHz XDBuses, with a peak sustainable read/write throughput of 500Mbytes per second.

500Mbytes per second may sound high, but each SuperSPARC processor is capable of executing three instructions per cycle, including supplying 64 bits per cycle from memory, so a 20-processor SC2000 could conceivably request 8 *billion* bytes per second from the memory system. Hence two layers of caches.

The first level, being on-chip, can react with new data in one cycle. The second level takes 5-10 cycles, whereas accessing the main memory takes 20-60 cycles. A high cache hit rate is therefore crucial to good performance.

A higher cache hit rate can be achieved if *false sharing* is avoided. False sharing occurs in a shared-memory multiprocessor when two processors access different data structures that happens to be allocated in the same cache line. It is therefore important to separate data structures that will be accessed by many workers into separate cache lines. This might mean that the local cache in a uni-processor system is not fully utilized.

8.1.2 Threads

The Solaris Multithreading system is used to implement the system [38]. The system allows a Unix process to be divided into multiple threads. Each thread has its own program counter, registers and execution stack but they all share the the same address space. The shared address space allow threads to easily access and modify a shared data representation. This is of course possible to achieve even in a multi process system but the multithreading library makes it much easier.

Another advantage of the multithreading system is that one can assign a set of LWP's to a process. A LWP is scheduled by the Solaris operation system. The more LWP's that are assigned to a process the more threads can execute in parallel. The threads in a process are scheduled fairly on the available LWP's and the LWP's of all processes are scheduled by the operating system. There is no need in allocating more LWP's than there are threads in a process. It is however perfectly all right to create more threads than there are LWP's.

In the Penny system each worker is implemented as a thread that is bound to a unique LWP. By binding the thread to a LWP we avoid the internal scheduling of threads among available LWP. We also ensures that the threads can be executed in parallel if there are enough processors available.

It is possible to bind a LWP to a processor or having the LWP's scheduled by a non-preemptive real-time scheduler but both of these solutions require that the system is executed in supervisor mode. If the system is unloaded there is little to gain in using these options, the Solaris operating system does a good job in keeping the LWP's on the same processors. If the system is loaded, it will of course be a great advantage to use the real-time scheduler. The question is of course whether you survive the social pressure or not.

8.1.3 Tools

The Gnu C compiler (gcc version 2.7) has been used in the development of the system. The compiler allows program labels to be treated as data. This feature is used to implement a threaded code emulator. The performance decreases with about 20% if the system uses a byte code emulator. In order to port the system to C compilers that do not provide this feature a byte code version can be compiled by setting a compile-time flag.

The GNU gdb debugger has been used for debugging. This debugger is not easy to handle since it does not know about the multiple threads. There is no support for following the execution of one thread or switching over to another thread. There are now tools available for debugging of multithreaded programs but these tool were not available when the system was implemented.

In the last phase of the project the instruction level simulator SimICS [43, 42, 44] was used. The simulator accurately simulates a multiprocessor SPARC architecture and can provide valuable statistics. The simulator was mainly used for performance debugging and proved to be a vital tool.

The simulator does provide a feature that could have saved many days (not to say weeks) of debugging. Since the simulator is a deterministic program, it is possible to redo the same execution twice. This is almost impossible to achieve when running the system live. If an execution generates an error the bug can be traced in much the same way as a bug in a sequential system.

8.1.4 Locks

We will use an atomic-swap operation on a single word to implement locking primitives. The locks will not hold simple locked/unlocked values but rather hold values as for example locked/dead/pointer. All locks will be spin-locks, i.e. a worker will swap the value of the lock until the lock is taken. This does of course not guarantee progress of each worker but it does not matter as long as the computation as a whole makes progress.

The swap locks are used instead of the locks provided by the Solaris threads library to conserve space. A Solaris "mutex_t" lock uses 24 bytes. It would induce a large overhead if these locks were used to protect single pointers. The swap locks are integrated in the pointer they protect and does not induce any space overhead. This especially important for the representations of terms where variable can be represented by a single word.

To get an idea how often locks are missed, SimICS *counters* were placed around the lock primitives. Counters are added by modifying the source code i.e. by adding SimICS macros that will turn the counters on and off. When running the Smith benchmark with sixteen workers, almost eighty-thousand variables were examined by a worker resulting in more than forty-thousand lock (xmem) operations. In only

REF	a reference term
UVA	an unconstrained variable
GVA	a constrained variable
LST	a list
STR	a structure
GEN	a generic structure
INT	a small signed integer
ATM	an atom

Table 8.1: Primary terms

one case did a collision occur. These figures indicate that the variable locks in the Penny machinery are not a performance bottleneck. In fact, it might be worthwhile redesigning some of these locks to be more aggressive in assuming low contention.

There is more contention on locks that protect stacks especially when there are few tasks in the system and too many workers. A conflict rate of as much as 10% has been measured.

8.2 Terms

Terms are implemented using tagged pointers. There are eight types of primary tags, these are listen in Table 8.1.

A list-pointer point to a cons cell. A structure pointer points to a structure holding a functor and a tuple of arguments. The functor is represented as a pointer to a structure holding the name, an atom, and arity of the functor. A generic structure-pointer points to a type specific structure were the first field determines the type.

Cons cells and structures have an optional environment identifier field. This field could be used during unification and copying but is not strictly necessary. The optional field does not cause any measurable overhead in the system as it is implemented today but could make a large difference in a highly optimized system especially for programs that produce a lot of list cells.

An atom contains a pointer to a print name and a saved hash value. The hash value is used to avoid rehashing when looking up definitions. The definition hash table is only used when programs are loaded. The print name is only used when an atom is printed. In all other parts of the system an atom is simply an identifier.

The integers are limited to the range $+/-2^{25}$ which is not much, but since larger numbers are handled by the GNU bignum package it does not cause a problem. The bignum package is however itself a bottleneck in the system. A global function is used to allocate new bignum structures. The function cannot be given any parameters so the only way to control where the structures are allocated is by replacing the function with a worker specific function. A global lock must therefore be taken each time a bignum operation is performed. There is of course no reason for the bignum package to be implemented like this and it will probably change once it is adapted to multithreaded applications.

8.2.1 Generic structures

Generic structures are used to implement for example floats and bignums but could be used for any kind of structure that one wants to handle in the system. A generic structure holds a pointer to a tuple holding at least one field. This field is a pointer to a method table that is unique for each type of generic structure. Since the method table is unique for each type the pointer can be used as a type identifier.

The method table holds a set of pointers to C procedures:

unify called on unification,

print produces a string representation of the structure and

new/copy/gc/deallocate are used by the garbage collector and copy procedures.

After the method pointer, comes a sequence of type specific data. The only parts of the system that knows about the internal representation of the structure are the procedures in the method table and built-ins that have been added to create and access the specific term. This makes it easy to add new data structures into the system. The method table makes it self contained.

Generic structures can contain normal terms and even pointers to structures that have the term itself as a member. The circular references can be handled both by the garbage collector, the copying procedure and the unifier.

Floats and bignums could be implemented using the normal generic structure protocol but they are instead represented by unique tags instead of method pointers. The system therefore must know how to handle these types. This is done since these types are heavily used and have to be handled by the normal arithmetic built-ins.

8.2.2 Variables

In the previous chapter the general representation of variables was presented. Variables were divided into unconstrained and constrained variables. In the implementation we make a distinction between three types of variables: unconstrained, generic and term variables.

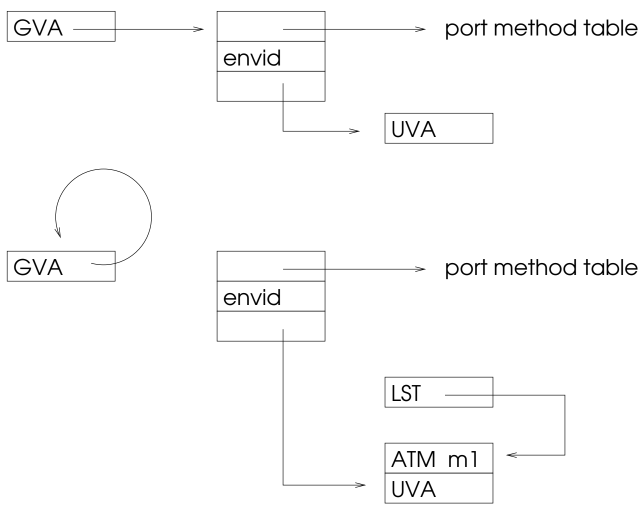

Figure 8.1: Sending a message to a port

Unconstrained variables are tagged with `UVA` and hold, as explained before, a pointer to the environment identifier of the variable. The variables tagged with `GVA` hold a pointer to a separate structure containing at least two fields. The first field is a pointer to a method table or a null pointer and the second one is a pointer to the environment identifier.

If the first field holds a null pointer it is a term variable. The structure then contains one extra field, the list of suspensions. If the first field contains a method pointer the variable is called a *generic variable* and is used to represent port like structures.

8.2.3 Generic variables

Generic variables differ from generic structure in two ways; the variables always holds a pointer to an environment identifier and the method table contains one additional procedure, the *send procedure*. The environment identifier is needed to determine if the generic variable is local or external to the current and-node when a message is sent. Only if the structure is local to the current and-node will a send operation be admissible.

Generic variables were first used to implement ports. A port is represented as a generic variable with one extra field, the *stream*. The stream is a term that should be unified with a cons cell when a new message is sent to the port. Figure 8.1 shows the representation of a port structure and how a send operation changes the structure. The GVA cell is first locked, this gives the worker exclusive rights to update the structure. A new cons cell is built with the message in the car field and a new variable in the cdr field. The stream, in this case an unbound variable, is then unified with the list. The stream pointer is updated to point to the new variable and the GVA cell is unlocked.

REF	`0............................00`
UVA	`0............................01`
CVA	`0............................11`
LST	`0..........................0010`
STR	`0..........................1010`
GEN	`0..........................1110`
INT	`0.........................00110`
ATM	`0.........................10110`

Table 8.2: Tagging scheme

Notice that the send operation is a destructive operation but no lock field is required in the port itself. The lock is taken when the variable cell is locked. Also notice that the stream is unified with the new list structure i.e. if the stream term was already bound to a list, the list would be unified with the new list.

The generic variable was first used to implement ports but has also been used to implement different kinds of variables, for example finite-domain variables, and port-like objects such as arrays and hash tables.

8.2.4 Tags

The tagging scheme uses the least significant bits. The two least significant bits are primary tags to distinguish REF, UVA and CVA from other tags. The other tags use the next two bits to distinguish LST, STR, and GEN from the INT and ATM tags. In addition to the normal tag bits the most significant bit is reserved for the garbage collector. Table 8.2 shows the used tag scheme.

The tagging scheme is efficient but has some drawbacks. Since the structure pointers are limited to 27 bits an address space of only 2^{29}bytes (512Mbytes) can be addressable. This is equivalent to 128M terms or 64M list cells, a big but by no means an enormous figure. An early version of the system had no detection of when the address space was exhausted. Some hard to find bugs occurred when the heap overflowed and started to write in random places.

The scheme favors operations on variables and reference pointers. These will not need a shift operation. A tag scheme with three primary tags for all terms might have been a better approach even if this would induce an extra shift operation when variables and reference pointers are accessed.

The usage of the most significant bit as a garbage collection bit has caused some problems especially when the system was ported. The garbage collector requires that every term could be marked so it is not sufficient with a single garbage collection tag. The high end bit was chosen to make the system more independent of the garbage collector.

8.3 Copying

A stable and-node is eligible for choice splitting. The operation is implemented by making a copy of the subtree formed by the candidate's parent and-node. The siblings of the candidate are not copied and the candidate is removed from the original subtree.

When the subtree is copied new instances must be made of all variables that are local to the subtree. Variables that are external to the subtree should be shared by the two copies.

8.3.1 Local and external structures

Two things are important to understand in order to make an efficient copying procedure. The first thing is that a structure that is external to the the subtree cannot hold any references to variables that are local to the subtree. The second thing is that no other worker has access to a local structure.

A structure that is created in an and-node can only be filled with terms that are accessible in the and-node i.e. local variables, external variables, terms created in the and-node and terms that have been created above the and-node. It can never contain data structures that have been created in a sub-node of the and-node unless this node has been promoted. If we give an environment identifier to each structure we create, we can recognize a structure that is external to the sub-tree. If the structure is external it can also be shared by the sub-tree and the copy of the sub-tree since they should share all variables that possible reside inside the structure.

Since the copying procedure is used only inside a stable and-node we know that no other worker will enter the and-node until we have finished the procedure and added tasks to invite other workers. Since workers outside the and-node does not have access to the structures that have been created in and-nodes inside the sub-tree the copying procedure can freely modify the local structures. This is important in order to implement copying of circular structures efficiently.

8.3.2 Three phases

The copying procedure is a three-phase operation. In the first phase a copy of the tree is created but no terms are copied. And-nodes are marked as copied and given forward pointers to their copies. In the second phase terms are copied. If the environment identifier of a variable or structure refers to a marked and-node the term is local to the sub-tree and is also copied. The third phase resets all marked and-nodes and forward pointers. It also copies all suspension lists of the local variables.

A copy operation is performed by a worker independently of the execution in other parts of the configuration. No external structures are modified during the operation. The local structures are modified with forward pointers to the copy. The forward pointers allow us to copy circular structures.

Ground terms need not be copied. As soon as it is determined that a structure is ground (during copying or garbage collection) a null identifier can replace the original identifier. In subsequent copy operations the ground term can be shared. This optimization has however not been implemented in the system

The overhead of creating the environment identifier is large for smaller structures such as lists cells but the environment identifier in a structure is not strictly necessary. External variables and ground structures can be shared and a structure need only be copied if any of its components needs to be copied. The existence of the environment identifier makes the detection of external and ground structures more efficient.

8.3.3 Updating

In the last phase of the algorithm all forward pointers are reset. The forward pointers of the and-nodes can easily be reset since this only means a traversal of the sub-tree. The resetting of term forward pointers is more complex. Since the structures have been destroyed we need a value trail with pointers to all cells that should be reset. These trails can of course be large and a better solution is most welcome.

The updating phase also takes care of the copying of suspension lists. Local variables can have suspensions that must be copied. External variables can also hold suspensions that are referring to entries in the copied sub-tree. These suspensions must be duplicated. This is a rare situation that only happens if the copied and-node is not the stable and-node.

8.4 Memory management

The memory is divided into areas that are handled differently. The size of some data structures can be determined at compile time but for some structures the size is known first when the system is booted. Some areas have a fixed size once allocated while others can grow during runtime. The memory areas are divided into:

static Structures that have a fixed size defined at compile time. These structures are initialized at boot time and can be accessible to all workers, typically global registers, locks etc.

worker Structures that hold the state of each worker. Since the number of workers are determined at boot time these structures must be allocated at boot time. The stacks needed by each worker must be able to grow during run time.

free-list Structures used for and-nodes, choice-nodes, choice-continuations, and-continuations, and and-node constraint entries. These structures are explicitly reclaimed. The list is initialized during boot time but can grow during run time.

constant Program specific structures that need to be maintained throughout the execution. For example functors, atoms, procedure definitions and hash tables to locate such structures. The area is allocated during boot time but can grow during runtime. The area is not subjected to garbage collection.

heap Program specific structures that do not have to be maintained throughout the execution. This is the area for terms, suspensions, insertion cells, environment identifiers, and hangers. All structures that can not easily be reclaimed explicitly. The area is allocated during boot time but can grow during run time.

external Structures that could be allocated on the heap but for practical reasons are allocated outside of the heap. These structures are allocated separately and can also be freed during run time.

The static, worker and constant areas are all handled straight forward. For the implementer of an embedded system an interesting observation can be made. Since we do not have a meta-call primitive there is no need to maintain a hash table that maps procedure names to definitions once a program has been loaded. Furthermore, if we remove chars_to_atom/2 and list_to_term/2 from the language (or detect that they are not used) there is no need to keep the hash-tables that maps strings to functors or atoms. The constant area would then be fixed in size during an execution.

8.4.1 The free-list

The free-list is allocated when the system is initialized. Each worker then receives a number of blocks that it will add to a private free-list. If a worker runs out of blocks it can request more blocks from the global free list. If the global list is empty new blocks are allocated by the operating system.

Since a worker will not necessary reclaim its own blocks a worker can build up a large list of blocks in its own private free-list. To prevent this the workers will return all but a certain amount of blocks to the global pool of blocks when a certain limit has been reached. The limit is by default set to one thousand blocks and one hundred blocks will be kept. The worker keeps a pointer to the last block to easily add the sequence of blocks to the global free list. Without this balancing of available blocks the system can break down as more and more blocks are allocated from the operating system.

The total number of blocks that is needed is of course depending on the program. Programs that create a lot of suspensions can easily use hundred-thousand blocks while non suspending programs can manage with a handful. The default is twenty-thousand blocks.

8.4.2 Reclaiming nodes

By reclaiming nodes explicitly we will decrease the garbage collection time and more importantly improve cache performance. In order to make the best possible use of the reclaimed structures we would like all nodes to use the same kind of blocks. If all structures could use the same blocks much would be gained. It is of course easier to maintain one free-list but it will also improve cache performance. For example, if the memory used by a reclaimed and-node immediately can be used for a constructed choice-node, this memory segment will most certainly be in the cache.

Nodes can be reclaimed in different situations:

failure A worker that fails an and-node can reclaim the and-node if it is alone in the and-node. The node is otherwise reclaimed by the last worker to leave the node.

pruning A worker that prunes an and-node can if the node is deserted reclaim the node. The node is otherwise reclaimed by the last worker to leave the node.

promotion A worker that promotes an and-node can reclaim the node.

last The worker that reclaims the last and-node under a choice-node can, under condition that the choice-continuation is empty, also reclaim the choice-node.

deallocate The deallocate instruction will reclaim the current and-continuation.

wake A wake task will reclaim the woken entry.

recall A recall task will reclaim the recalled choice-node.

A choice-node or an and-continuation can easily be reclaimed. There are no references left to either. This is guaranteed by the single reference property of the and-continuation and the fact that at most one variable can have a reference to a suspended goal.

When an and-node is reclaimed the body, all entries of the and-node and all nodes below the and-node must be reclaimed. This could be a costly operation but will not be expensive in practice. When a choice-node is reclaimed the choice-continuation is also reclaimed.

Notice that if it is decided that a node should be reclaimed there will not be any installed workers in the sub-tree formed by the reclaimed node. There might however be living references to both the environment identifier and hangers of the entries of a reclaimed and-node. The and-node must therefore be locked and for each entry that is reclaimed the hanger must be killed.

8.4.3 The size of a block

Choice-nodes, and-nodes and entries have different size but they all have a fixed size and can thus easily be maintained in one free-list. And-continuations and choice-continuations can have any size and if these are to be reclaimed we have different choices. One possibility is to break the continuations up into fixed size segments. This has the drawback that it is harder to access the continuations since the registers are not in consecutive order.

Access by instructions to the and-continuations is most important to implement efficiently. Since all accesses by instructions are known at compile time the compiler can be made aware of the segmented representation and issue special instructions for the access that are not in the first segment. If these accesses are infrequent the only drawback is that the number of instructions is increased. This can be avoided by adding only two new instructions: one that will set the and-continuation register pointer to the right segment and another that will reset it. This will of course slow down the access to registers not in the first segment even more, but it could pay off depending on the technique used in the implementation of the instruction handler. A native code compiler would of course issue the correct sequence of instructions but in an emulated system there is a benefit of having a small set of instructions.

In the current implementation the and-continuation is divided into blocks. There is no compiler support for efficient access to an and-continuation, no special instructions are used to access the registers of the and-continuations.

We will of course lose some space if we use the size of the largest structure to represent even the smallest structure but this is not so important. The size of a cache-line of the second level cache on our target machine, is 64 bytes. Choice-nodes, and-nodes and entries all fit into 64 bytes. If the continuations are broken up into 64 byte segments each segment will hold a maximum of 15 register. The first segments will hold fewer (choice-continuation 13, and-continuation 10) since these segments also hold other pointers. We have however found that this is not a serious limitation but a careful study of source code and executions is of course necessary to verify this.

8.4.4 The Heap

Even the heap is divided into blocks. How many blocks that are created and the size of the blocks is given as a command line argument. A default value is thirty-two blocks each of 8Kbytes. This is sufficient to run even larger programs such as the compiler. The size of the blocks does of course limit the size of the largest possible term but there is nothing that prevents that bigger blocks are allocated during runtime.

The heap blocks need not be ordered in any way. The address of terms are never used for comparison[1]. This allows us to mix the blocks in an arbitrary order and request new blocks during run time.

The blocks are held in a list and are requested by workers as needed. A block that is given to a worker is moved from the list if unused blocks to a list of used blocks. As the number of used blocks exceeds a certain threshold, given as a command line argument, the garbage collection flag is set.

A worker need only keep track of its current block. When it allocates structures in the block it always checks the boundary of the block. There is no optimizations implemented that will reduce the number of checks, this is of course possible with compile time analyze.

8.4.5 External data

The generic structures allow us to allocate and handle arbitrary structures allocated external to the heap. For each type of structure a unique generic structure type is defined. A generic structure consist of a method table and a sequence of type specific data. There is nothing that prevents that the sequence contain pointers to data structures that are allocated outside of the heap. The only problem is that if the program looses the reference to the generic structure the externally allocated data structure might not be freed properly.

To solve this, all generic structures that contain references to externally allocated memory must add themselves to a list when they are created. After each garbage collection this list is traversed. Structures that have been copied remain in the list but structures that have not been copied, and subsequently are dead, get a chance to deallocate externally allocated structure.

The scheme is used also for external resources such as open files. If a generic file structure is found un-copied in the list of generic structures, the file is closed. The scheme is flexible and has proved most useful.

8.5 Garbage collection

The garbage collector was designed and implemented together with Galal Atlam and Khayri Ali. Both the strategy, its implementation, and performance has been described in the thesis by Galal [5], this section will only be a brief overview.

[1] Except during unification. This order need not be preserved.

Parallel garbage collectors have a long history [27, 6, 20, 60]. The one that come closest to our implementation is the implementation by Imai and Tick [30]. They parallelized a stop-and-copy garbage collection scheme for shared-memory multiprocessors. The system was implemented within a concurrent logic programming system called VPIM, a parallel KL1 emulator.

The approach we have taken is a parallelization of a sequential stop-and-copy garbage collection scheme based on traversing active data in a depth-first manner proposed by Ali[3].

8.5.1 Stop and copy

The garbage collector is a *stop-and-copy* collector i.e. execution stops, the living data structures are copied to a new memory area and the old memory area is reclaimed [13].

Copying garbage collector is attractive for systems that have a high rate of garbage generation, as in functional and logical programming languages. The garbage collection time is proportional to the amount of data in use by the system [50]. This is in contrast to a mark-and-sweep technique, which has garbage collection time proportional to the entire area.

The one disadvantage with a copying collector is that it normally only can use half of the available space. An implementation often uses two equally big areas and copies from one area to the other. We can avoid this since we are working with a segmented heap. We can for example configure the system to use, for example eighty percent of the available blocks before garbage collection is performed. This is not a unrealistic figure since the living data is normally much smaller than the area used. If this is not the case, any system will have serious performance problems.

When a worker requests a new heap block the total number of living blocks is incremented. When the number of living blocks reaches a limit the garbage collection flag is set. The flag is periodically checked by each worker to detect when garbage collection is needed. The check is performed at each call, execute, proceed or suspend instruction and at entrance to the task or guard handler. The flag is also checked by workers in the scheduler.

8.5.2 Dividing the job

If a worker detects that the garbage collection flag is set it updates its execution state and saves any pointers that it has temporary cached in local registers. The worker then calls the garbage collection procedure and suspends on a barrier synchronization.

One worker, the *master*, manages the garbage collection process by signaling all other workers, the *slaves*. The master waits for all the slaves to report at the barrier.

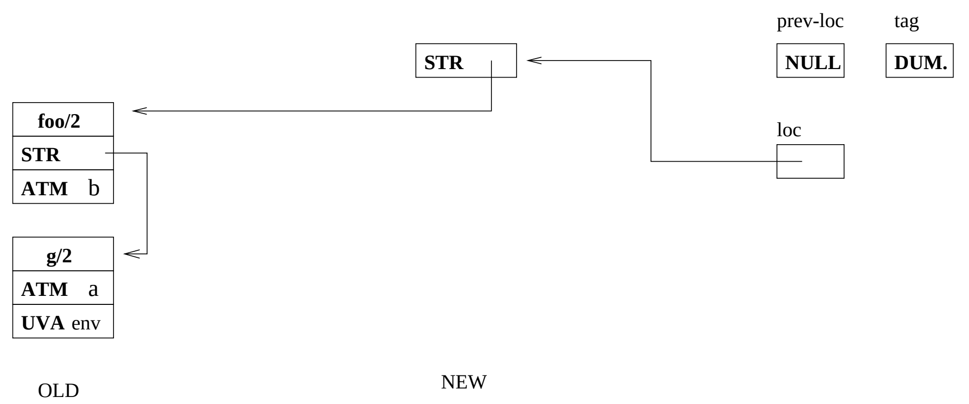

Figure 8.2: The initial setting

It then traverses the execution state and divides the and-nodes choice-nodes evenly among all workers. The and-continuations are always referred to form a continuation stack and it is best if each worker copies the continuations of its own stacks. When this is done the master signals to all the workers to start the garbage collection.

Since almost all data structures that make up the execution state uses free blocks, it is only the structures that are allocated on the heap i.e. terms, suspensions, insertion cells, environment identifiers, and hangers, that needs to be copied.

The workers now have several accesses to the execution state: the argument registers, the continuation stack, the wake stack and the stack of nodes that it has been allocated. It will start to copy all data structures that are accessible from these structures.

8.5.3 Scavenger

The terms are copied using a scavenger scheme. The general idea is to make a copy of a structure and then recursively copy the components of the structure. When the whole structure has been copied the pointer to the structure is updated. This is complicated and is best explained with an example.

Assume a term structure "foo(g(a,X),b)" that should be copied. The structure is represented by a tagged pointer in the new space pointing to the structure still in the old space. The algorithm works with three extra registers, the *loc* pointer that is pointing to the cell that should be copied, the *prev-loc* pointer that is pointing to the previous structure, and the *tag* register that holds the tag of the structure that we are currently copying.

Figure 8.2 shows the initial setting. The following steps will then copy the structure:

Step 1: Copy the structure pointed to by the loc pointer to the new space. Replace the first cell in the old structure with a marked forward pointer. Mark the last cell to be scanned in the new copy (the first argument). Push the copied structure pointer on the scavenger stack, that is, prev-loc and tag are saved

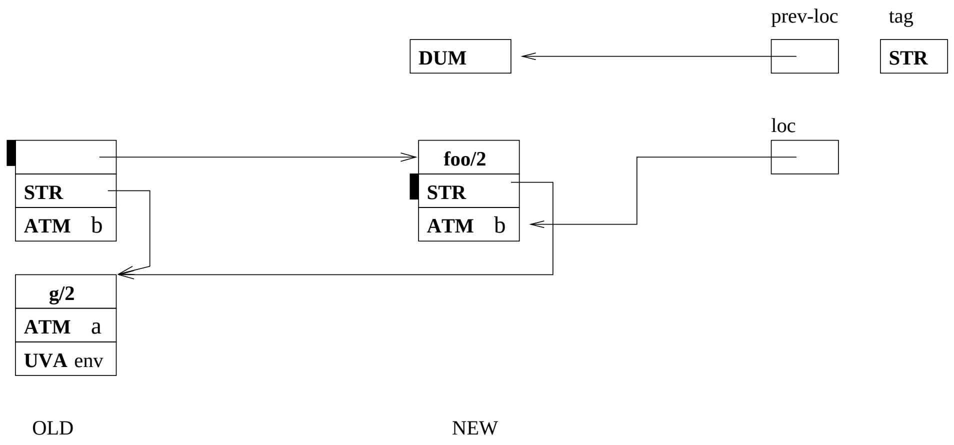

Figure 8.3: Copy the structure "f(_,b)"

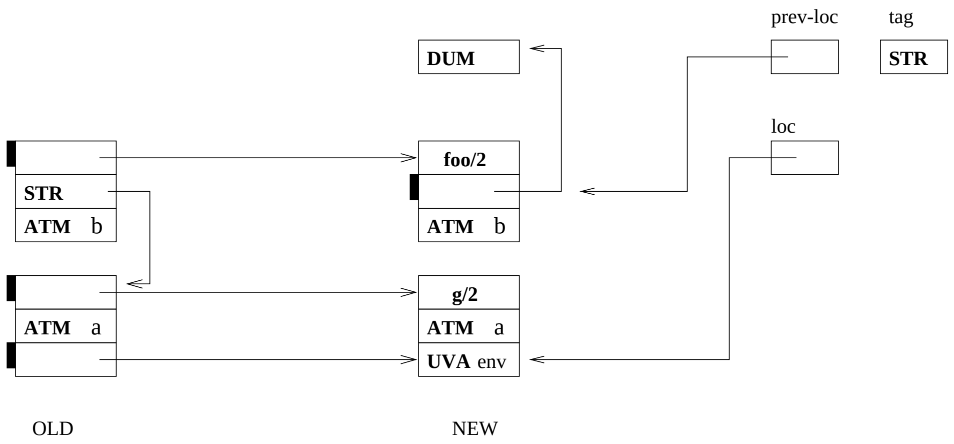

Figure 8.4: Copy the structure "g(a,X)"

into the cell pointed to by loc and the address and tag of the pointer is saved in prev-loc and tag registers. The loc register is then set to point to the first cell be scanned in the new structure. This is illustrated in Figure 8.3.

Step 2: The content of the cell pointed to by loc holds an atom, loc is decremented to point to the next cell to be scanned.

Step 3: The current cell to be scanned is a tagged pointer to the structure "g(a,X)". The first step is repeated and the result is shown in Figure 8.4. Notice that we also have to make a forward reference for the variable. The envid of the variable is dereferenced and stored in the new copy.

Step 4: The cell to be scanned is an unbound variable, the loc is decremented to point to the next cell to be scanned.

Step 5: The cell be scanned is an atom that is marked as the last cell to be scanned. The structure is copied and only the updating of the pointer to this structure remains. The prev-loc register holds a pointer to the cell where the structure pointer should be restored and the tag register holds the correct tag. The new values of the prev-loc and tag registers are fetched from the location of the

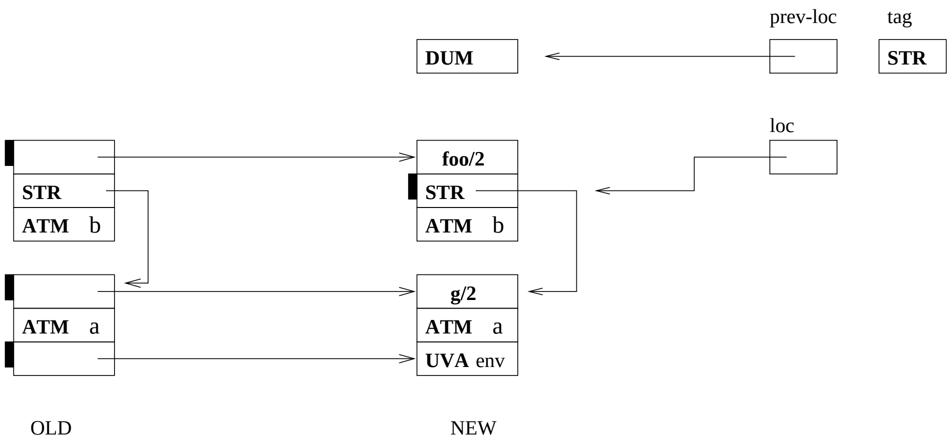

Figure 8.5: Updating the structure pointer

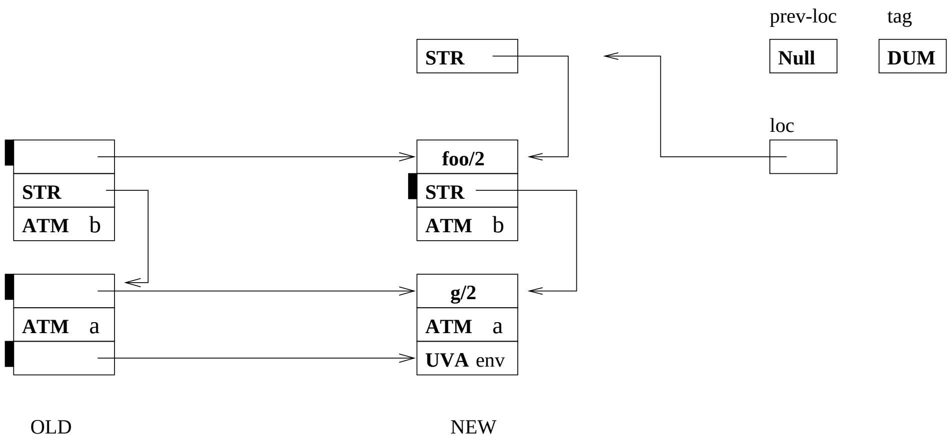

Figure 8.6: The final result

restored pointer. Figure 8.5 shows the situation when all registers have been updated.

Step(6): The loc register points to a marked cell, the last cell in the current structure. The same procedure as in step 5 is repeated resulting in the situation shown in Figure 8.6. The scavenger stack is now empty and the structure has been copied.

The forward marking of cells in the old space allows circular structures to be copied and the in-place backwards linking of structures does not require any extra space. A problem does however arise when a reference pointer to a variable is encountered. If this happens we do not know if the variable belongs to a larger structure. If we ignore this and and copy the variable right away the copy of the structure will contain a reference pointer to a variable instead of an in-line variable.

To avoid this all reference pointers are saved on a stack and processed when all other structures have been copied. If the variable is copied it belonged to a structure and we will find it using the forward reference. If the variable is uncopied it was a free variable and most be copied to the new space.

The reference stack is not attractive. Having a data structures that grows during garbage collection is asking for trouble. We do the garbage collection since we are short on memory, it is not the right time to allocate a larger stack.

The scheme could be changed so that reference pointers in structures, that are pointing to a variable, are collapsed. Each such operation will reduce the number of reference pointers by one. If the variable was part of a structure, the copy if the structure will contain an extra reference pointer but then we only add one. The copying of reference pointers in nodes (and-continuations, choice-continuations etc) that are not allowed to hold in-line variables is delayed until all structures have been copied. This will of course mean that the execution state is traversed twice.

8.5.4 Parallelizing the Scheme

In the sequential scheme, cells to be scanned are maintained by a contiguous sequence of cells in the current object, pointed to by loc, and a chain for all remaining cells to be scanned, pointed to by prev-loc. Ali, in his paper [3] proposes the idea for parallelizing the scanning of cells, by dividing the latter chain into a number of smaller sub-chains and to make them available to the other workers. Each chain is a piece of work that can be processed by any worker.

Figure 8.7 illustrates the idea of dividing the chain of un-scanned cells into segments. Each segment is represented by two pointers, the the *pi* and *li* pointers. The pi pointer is the holds the prev-loc information and the li pointer holds both the loc pointer and the tag value. The pointers are stored on a *chain-work-stack*, one for each worker. Idle workers can steal work from this stack by locking the stack and removing a pi-li pair. values.

The question is, how often the chains should be divided. A too large granularity will not divide the work among the workers while a too small granularity will create an overhead in the scavenger algorithm. A fixed threshold is set at compile time. The threshold does not change depending on the load or the number of workers.

8.5.5 A list of atoms

The worst possible scenario for the parallel scheme is the copying of a list of atomic terms. The copying time will be bounded by the time it takes to copy the list template to the new heap. This is a sequential process that is not parallelized by the current scheme.

The scheme works well for large, well balanced, structures but it has its worst behavior when encountering a list. The importance of this drawback was not realized until the performance was measured for actual programs. Long lists of atoms or integers are frequent and sets a limit on the obtainable speedup.

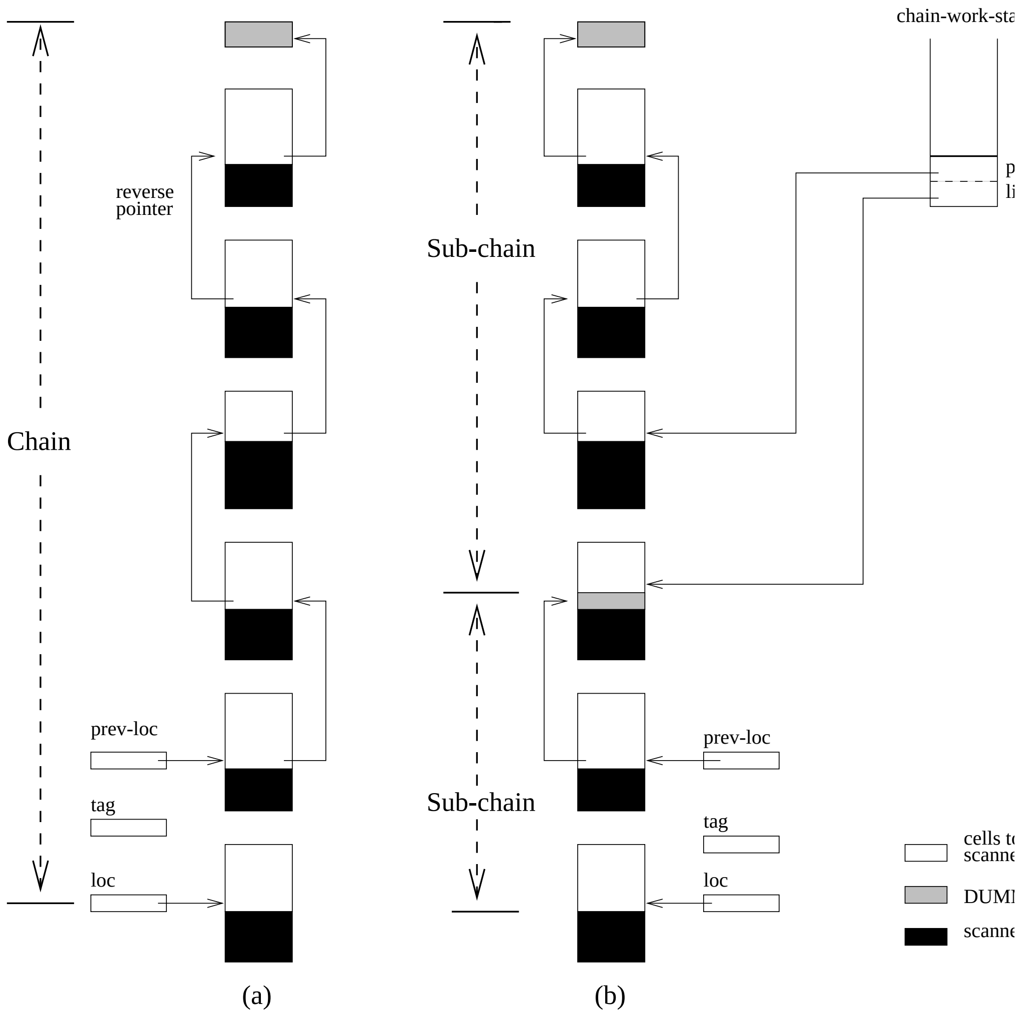

Figure 8.7: Dividing a chain (a) into segments (b)

The only solution to this problem is to divide the structure on the old heap before it is being copied. The sequential operation is still proportional to the length of the list but the actual copying can be done in parallel.

8.5.6 Generic structures

The garbage collector knows little about generic structures. When a generic structure is encountered the method table is used to copy the structure. There are two functions in the table that are used: the *new* and *gc* functions. The new function will create a template of the generic structure but will not copy its contents. The worker will then forward mark the original structure to the copy. When this is done the worker will call the gc function giving the original and template as arguments.

The gc function can now copy its data, using if necessary a *gc-term* function for any AKL terms. The sub-structures can be copied even if they contain a circular structure that refers to the generic structure. Data structures that do not reside on the heap need not be copied.

When all structure have been copied the master traverses s list of generic structures that have requested to know when they become garbage. If the master finds an uncopied structure its *deallocate* function is used to give it a chance to free any externally allocated structures or close open files etc. There is no auto-closing of ports like in the AGENTS system.

8.6 The emulator

The emulator is implemented using threaded code. This means that each abstract machine instruction is represented by a code pointer followed by a set of arguments. Each instruction begins by reading the code pointer of the next instruction so that the read operation can overlap with the operations in the instructions. This sound like an unnecessary optimization but does actually pay of remarkably well.

The instruction size varies between one word for instructions without any arguments to a program dependent number for the switch instructions. Indexes to argument registers are 16 bits which will be sufficient for all practical applications.

8.6.1 Instructions

The implementation of the instructions is straight forward. I will here only describe the most important differences between the implementation of the instructions in the Penny system and the sequential AGENTS implementation.

The guard instructions and procedural instructions are similar in their implementation, the most important differences are found in the implementation of the term instructions. The term instructions are, as described in Chapter 6, divided into put- unify- and get instructions. The put instructions will be implemented in the same way as in a sequential system and so will the unify instructions that execute in write-mode. The only difference will be in the get instructions and unify instructions that are executing in read-mode.

To understand how different the implementation is we will take a detailed look at the get_list instruction.

The first operation that has to be performed is to determine the type of the term in the argument register. It is either a reference, a list pointer or some other term. In case it is a list pointer or another term the procedure is the same as it would be in a sequential system. Only if the argument register holds a reference does the implementation differ.

If the register holds a reference we presume that it is a reference to a cell holding a variable. This must not be the case but it is likely that it is. An atomic-swap in-

```
{ Xi is the argument register;
  while( Xi.tag == REF )
    swap(Xi.value, Cont,  Xi);
    If( Cont.tag == UVA || Cont.tag == GVA) {
      bind the variable;
    }
    if( Cont != Xi ) {
      *Xi.value = Cont;
      Xi = Cont;
    }
  }
  if( Xi.tag == LST ) {
    save structure pointer;
    execute the next instruction;
  }
  goto the fail handler;
}
```

Figure 8.8: The code from the get_list instruction

struction will replace the contents of the cell with a self-reference before the contents is examined. The contents is either:

a variable , in which case the variable has been successfully locked,

a reference to the cell , in which case someone else holds the lock or

something else , in which case the value is restored but also placed in the argument register.

In case we have locked a variable we will continue with the binding procedures. If the variable is locked by someone else the atomic-swap operation is repeated. If the contents is something else the procedure will repeat itself.

An outline, in a C like syntax,of the code is listed in figure 8.8. The tag and value of a term `Xi` will be accessed by `Xi.tag` and `Xi.value`. The `swap(Xi.value, Cont, Xi)` statement will assign `Cont` the contents of the memory cell pointed to by `Xi.value` and write back `Xi`.

In case we have locked a variable we first check whether the variable is local or external to the current and-node. If it is external we must search for a local binding according to the binding scheme. If no binding is found or if the variable is local it should be bound to a list cell. The way this is done in the AGENTS system is that a list pointer, pointing to the top of the heap, is placed in the variable cell. The actual list cell is then constructed by the following unify instructions. This would not work in the Penny system. Once the variable is bound, another worker could access the list before the unify instructions have initialized the “car” and “cdr” positions. If we initialize the positions with locked variables (self-references) the worker will loop until the proper values arrive. The unify instructions that operate in write mode will ignore the contents of the positions and operate just like their AGENTS counterparts.

8.6.2 How expensive is it?

To understand how expensive the get-list instruction is we will compare it to the get-list instruction in the WAM [72]. We will therefore only consider the case were variables are local and unconstrained since this will be the case if we execute Prolog like programs.

If the argument register holds anything other than a reference term only the tag will be examined. This is the same as for the WAM. If the argument register holds a reference term an atomic-swap operation is executed. The WAM must here perform a read operation. In both implementation is the operation immediately followed by an instruction that examines the tag.

The atomic-swap instruction is normally considered to be an expensive operation but what few realize is that so is a read operation. Or rather the fact that the next instruction will stall until the value has arrived. There is nothing the processor can do wile waiting for the value and this can take a long time. If the value is not in the second level cache the difference between a read instruction and an atomic- swap instruction is not large. The difference is only significant when there is a shared copy of the value in the cache. In this situation the read operation can proceed without delay whereas the swap operation will have to invalidate other copies of the same cache line.

Once the value has arrived the tag is examined. Assuming that it hold a variable tag the environment identifier is examined to determine if the variable is local to the current and-node .this can be compared to determining the age of a variable in WAM. The WAM can then continue directly but the Penny system has to construct a list template on the heap. Both systems then write a tagged list pointer to the variable and continue with the next instruction. The two extra write operations are not expensive, a processor normally has a buffer with outstanding write operations and does not have to wait for the operations to be completed. The overhead of the extra write instructions is small compared to the cost of the read instruction.

This means that when a Prolog like program i.e. only local variables, the overhead is small compared to a WAM implementation of Prolog. If more than one processor is used an additional price is paid if a variable turns out to have a list of suspensions attached to it. If this is the case the lock operation has already been performed. The most expensive operation is then to read the suspension list. This is an expensive operation not because the list tends to be long (almost always one element) but because reading the list almost certainly will cause a miss in the second level cache.

The suspension was most probably added to the variable by another processor so this processor has an exclusive copy of the corresponding cache line and all other copies have been invalidated.

The extra price one has to pay in a parallel implementation is not so much related to a more complicated implementation or conflicts over locks. It's more related to the decreased cache performance.

8.7 Scheduling

Scheduling in parallel logic programming systems can be divided into two categories. In the first category we find schedulers that handles or-parallelism, the other category deal with and-parallelism. The two categories are different.

In an or-parallel Prolog implementation it is essential that the workers pick tasks that will explore the left part of the search tree in order to avoid doing too much speculative work. Since the semantic of Prolog require that the leftmost solution is presented first, any work done to the right of this solution could turn out to be useless. At the same time one must do speculative work in order to get any speedup from the parallel execution.

In an and-parallel system the question of which tasks that are executed is not as important. All task must be executed sometime unless the computation fails. The question is more how task should be evenly distributed over the available processors. In some systems the distribution is under the programmers control [14] while others will make the distribution automatically either statically at compile time or dynamic during run time. In the Penny system we wanted to experiment with dynamic scheduling without any user annotations.

In flat committed choice languages and in and-parallel or or-parallel Prolog implementations there is only one type of parallelism to consider. The Andorra-I system is one of the few implementations where research on load balancing for both and- and or-parallel work has been conducted [17]. The Andorra-I system is however limited to and-parallel execution in each leaf of an or-tree.

An AKL program differs from a Prolog or a KL1 program. We can have and-parallelism at all levels in the execution. The amount of parallelism could of course be restricted by only distribute work in the main and-node but this would severely limit the parallelism once the same computation was encapsulated in a deep guard.

The scheduler has been design to allow work to be shared on all levels in an execution tree. The scheduling procedure is thus complicated but the overhead is for the most common cases low.

8.7.1 The Stealer

The scheduler is design to minimize the overhead of a busy worker. The burden of the scheduling operation should as long as possible fall on the idle worker looking for job. This will ensure that the performance of a single worker execution is not decreased. The scheduler is also designed in a way that allows several tasks to be distributed in parallel, there is no central resource that will synchronize all scheduling operations. This will ensure that the system can scale well.

There are two requirements on a good scheduler. One is that a scheduling operation should be quick in order to get a fast distribution of tasks. The other is that a scheduling operation should be clever in order to minimize the number of scheduling operations that has to be performed. These requirements are often in conflicting with each other. We have after chosen to optimize the former.

The scheduler we designed is, in its principle design, simple. An idle worker will simply take one task from a busy worker and then continues the execution. Were little thought is spent on which task to take or from which worker to take it from. The scheduler was nick-named "the wild-west scheduler" for its brute approach but it turned out to be a good strategy. The scheduler was later renamed to "the stealer" and proved to be a hard to beat [69].

8.7.2 Task stacks

There are as described earlier three task stacks and one context stack. The stacks are protected by a single lock that must be taken to remove or change an entry. The owner of the stacks can add items to the stacks without having to take the lock. This is possible since the owner is the only one that can add anything to the stacks and workers that try to steal an item will steal them from the bottom of the stack.

A worker must take the stack lock when it removes a task from its own stacks and also when it need to update the an entry on the context stack. It would of course be possible to have one lock per stack to avoid contention but this would require more lock operations when a worker is looking through its local stacks.

The lock that protects the stacks is placed in a separate structure that is allocate in a block that is cache line aligned. Also the stack headers, containing the top and bottom of each stack is allocated in their own cache-lines. This is important since a steal operation by another worker otherwise could invalidate other important data that should be local to the worker.

Figure 8.9 shows how the stacks belonging to a worker are allocated. An array that is known by all workers hold an entry for each worker. This entry contain pointers to the lock and *stack headers* of each worker. The cache lines that hold information that only the the owner needs are never touched by another worker. This optimization proved to be important.

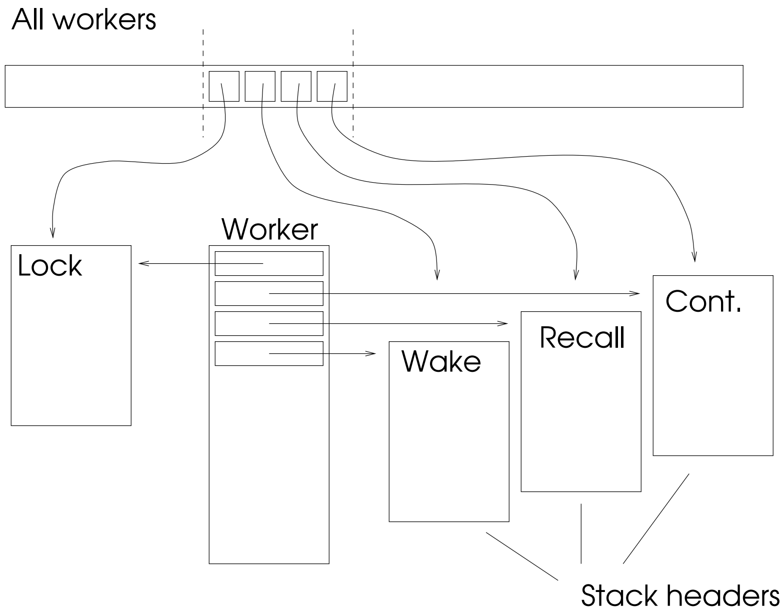

Figure 8.9: Representation of workers

8.7.3 Stealing a task

A worker that enters the scheduler registers as idle. The search procedure then begins by checking if all workers have registered as idle. If this is the case the execution terminates. Next the garbage collection flag is checked. The worker can contribute to the garbage collection although it does not have any private areas to start from.

The worker will then start the search for a new task. It will first look for a recall task, then a wake task and finally a continuation task. It will search all workers for one type of task before it moves to the next set of task. If no task is found the worker will start the search procedure from the beginning.

A worker that steals a task from another worker must first take the stack lock of the worker. It will then try to remove the last entry of the stack. If no entry is found the worker releases the lock and tries to steal a task form another worker.

If a task is found, not only the stack header but also the context stack entries of the busy worker must be updated. If the task did not belong to the main and-node there might be several context entries that must be updated. The same number of context entries must also be added to the stealing worker and the corresponding and-nodes must be entered. There is a dead-lock situation here that must be avoided. A busy worker must never hold a lock of an and-node while waiting for the stack lock. If it is being robbed it might have to wait forever.

Since the locking of the stacks prevents the owner from removing an entry the lock operation should be prevented if possible. A worker will therefor first scan a busy workers stacks in order to find a suitable task. If none is found, no lock operation is necessary.

8.7.4 Waiting

If all workers have been searched and no task is found the worker should start from the beginning. This is however an indication that there might not be enough tasks in the execution to keep all workers busy. To keep all idle workers in the scheduler might lead to poor performance (as will be shown in the next chapter). And dummy loop is therefore added at the end of each search phase to delay the workers.

If the system is functioning as a server that is waiting for its next request the idle workers should of course not be in an idle loop but be suspended. This is however tricky to solve since the workers must be woken up as quickly as available work turns up. This has not been implemented but must surely be addressed by a system that should be able to work as a process with long periods of inactivity.

8.7.5 Global pool

The scheduler also keeps a global queue of tasks. Tasks can be added to this pool by interrupt routines that are waiting for a timeouts or data arriving on a socket connection. This pool of tasks is checked in each iteration of the search procedure. The system also has to keep track of suspended interrupt processes so that the execution does not terminate if there are still task that could be added to the global pool.

We also experimented with schedulers that used the global pool to balance the available tasks. A worker could for example steal several ,task and leave some in the global pool or voluntary add tasks to the pool when its own tasks stacks grow above a limit. These schedulers all had their example programs where they excelled but they did not show as stable figures as the simple stealer.

8.8 Cache performance

SimICS emulates a cache hierarchy which is close enough to real life to give a good prediction of performance of an application. In fact, during our profiling we initially had significant discrepancies between predicted performance and measured values, until we discovered that both the SC2000 machines we used for timing measurements had faulty SuperSPARC processors with only 4Kbyte caches, not 16Kbyte. The faulty processors had gone unnoticed for several years, despite the machine being used extensively for benchmarking of parallel programs. Therefore, all simulated values in this paper assume a 4Kbyte first-level cache with 32-byte cache lines, and a 2Mbyte second-level cache with 64-byte cache lines, both direct-mapped.

In this section, we will use SimICS to study the Penny system form various perspectives. As input to Penny we have selected a small number of AKL programs, listed in table 8.8. These are used to exercise Penny with a variety of problems.

Application	**Description**
Life	The benchmark involves little computation, since the rules for determining each generation are simple. It is heavy on communication, however, since each object needs to communicate with all the neighbors. The benchmark simulates 40 generations in a 20x20 world.
Trace	The benchmark renders a small image of a mouse made out of spheres. Very little communication since the rays can be calculated independently but heavy on calculation.
Smith	The benchmark matches DNA sequences using the Smith–Waterman algorithm. This is a relatively simple algorithm, but it is hard to parallelize because the synchronization is fine-grain. The problem solved is matching two 400 elements long sequences against each other.

Table 8.3: Sample benchmarks used to exercise Penny

	Life		**Trace**		**Smith**	
	4 cpus	16 cpus	4 cpus	16 cpus	4 cpus	16 cpus
Exec. time (ms)	1301	739	2373	923	4142	2083
Instr. count	159754500	140831500	416811500	365338500	772478500	728961500
Instr. footprint	1296	1361	2123	2405	2377	2653
TLB misses	482783	710159	108140	231670	94289	314461
Read count	29944621	31322773	75663119	77990473	123426208	135877087
L1 miss rate	9.75%	9.54%	8.43%	8.92%	5.83%	5.74%
L2 miss rate	0.88%	1.66%	0.06%	0.13%	0.45%	0.55%
Write count	13245667	13430186	33281214	33555036	45725860	45882835
L1 miss rate	10.31%	11.97%	7.18%	8.00%	6.68%	6.92%
L2 miss rate	2.55%	4.93%	0.94%	1.17%	0.92%	1.26%

Table 8.4: Sample benchmarks, real and simulated characteristics

Though running on the same emulator, these programs trigger remarkably different behavior.

We begin by generating global statistics to characterize the input programs. In table 8.4, we've run SimICS on a base version of Penny, with the three benchmark programs and two sizes of machines, 4 or 16 processors. The first line in the table is execution time as measured on the actual target machine, defined as total runtime minus initialization. These timings have a standard deviation of approximately 1.5%. The remaining numbers are reported by SimICS running the same workloads. The numbers are the sum of of all CPUs. The instruction count is only based on sampling and is not an exact figure.

The figures are revealing. The instruction footprint is low. This is the number of individual machine instructions that have actually been executed. Out of the more than one forty-five-thousand lines of C code that makes up the system less than three-thousand assembler instructions were actually executed. This manifest the importance of optimizing the right part of the code.

The read and write counts stays more or less the same whether four or sixteen workers are used. The miss rate of the first level cache also stays the same, varying between 5% and 10%. The miss rate of the second level cache does increase for all benchmarks as the sixteen workers are used, this is significant for the performance of the system.

8.8.1 Deciding on prefetching

The get instructions will verify the type of the term. Subsequent unify instructions will access the arguments of the term. In the case of a simple list term this results in three instructions—one for verifying that the term is a list tag and two for accessing the "car" and "cdr" of the list cell.

We noticed an exceptionally high read miss rate in the unify instructions that accessed the components in a term. The reason was that when the instructions where used they were often reading terms that had been constructed by another worker. The terms had thus been constructed in one cache but where often read by another processor.

The remedy to this problem was to add a prefetch instruction (coded in C) in all get and unify instructions. The prefetch would read the next argument position so that the following unify instruction would find the value in the cache. The time to read the argument would hopefully overlap with useful work.

The fix did not work as expected. The read misses in the unify instructions did decrease but we had added a large number of read instructions, most of which contributed nothing since the data was already in the first-level cache.

The reason for this was obvious. When a functor or argument of a term was inspected the following argument would in seven cases out of eight be in the same cache line of the first level cache. The same would hold for any prefetch instruction in the unify instructions. The only place where a prefetch instruction would make any sense was in the get instruction that was responsible for verifying a list cell. The get instruction itself would only read the tagged pointer to the cell but not the car nor cdr of the cell.

Removing all but the one effective prefetch left us with an overall performance improvement of 3-4% for several of the workloads.

In this example, we were able to use SIMICS both to follow where the cache misses moved to, and to quickly quantify the overhead induced by the fix. Note that an optimizing compiler could not have identified this prefetch, since it wouldn't know about the restrictions placed on the sequences of abstract instructions—this required the intervention of the Penny designer, using SIMICS to explore trade-offs.

8.8.2 Read and write operations

Table 8.8.2 shows statistics gathered from executing the Smith benchmark on the improved Penny system. The reported numbers are from one of the CPUs. The cache statistics are for a 2Mbyte cache, i.e. our target machines second level cache.

	Number of workers				
	1	2	4	8	16
runtime (ms)	13822	7238	3996	2487	1789
read operations (x10^6)	124	59.5	29.8	15.2	7.74
read misses (x10^3)	8.66	121	71.7	43.6	29.0
miss rate	0.007%	0.204%	0.240%	0.287%	0.375%
write operations (x10^6)	47.5	22.3	11.1	5.60	2.81
write misses (x10^3)	138	143	80.9	52.8	34.0
miss rate	0.292%	0.643%	0.725%	0.942%	1.211%
Instr. count (x10^6)	610	308	156	81	42

Table 8.5: Profiling (of one CPU) and timing of Penny running Smith

The number of read and write operations is a good measure of the amount of "work" performed, and the figures in the table indicate an uncanny linearity in the load per processor, i.e. no significant overhead is added. Yet the speedup is not linear.

	number of workers			
	1-2	2-4	4-8	8-16
runtime	0.52	0.55	0.62	0.72
read operations	0.47	0.50	0.51	0.51
write operation	0.47	0.50	0.50	0.50
instruction count	0.50	0.51	0.52	0.52

Table 8.6: The improvement of operation count

Table 8.6 shows the improvements for every doubling of number of workers. The execution time is reduced to 52% as (speedup 1.9) when going from one to two workers. The efficiency then drops and moving from eight to sixteen workers only reduces the execution time to 72% (speedup 1.39).

The explanation for the decreasing speedup cannot be found in the number of read or write operations or the total number of instructions executed. As seen the number of read and write operations per processor (this is statistics from one of the processors) is more or less cut in two for every doubling of workers.

	number of workers			
	1-2	2-4	4-8	8-16
runtime	0.52	0.55	0.62	0.72
read misses	14.0	0.59	0.61	0.66
write misses	1.04	0.57	0.65	0.64

Table 8.7: The improvement of cache performance

The explanation is found in the number of read and write misses. Table 8.7 shows the same relative changes but for cache misses. When we go from one to two workers the number of read misses increases with a factor fourteen (!) and the number of write misses stays the same. At the same time the total number of read and write operations has been cut in half. This is the most likely explanation why the initial speedup is only 1.9 and not closer to 2.0 as indicated by the operation count.

The read and write misses does behave better once we use more than two processors but does not come close to the figures reported for the instruction counts. It should however be noted that the read and write misses are reduced to 66% or less while the execution time is reduced to 72%. The poor performance when using sixteen workers cannot be explained only by the number of read and write misses alone. This is investigated further in the next section.

In this analysis, SIMICS reports sufficient detail to help us reconstruct what is causing speedup to begin trickling off. We now know that for larger configurations, we should focus on second-level cache misses. Many of the data structures in Penny have been optimized to be cache-line aligned, etc. The SIMICS' source-line profiling of second-level cache misses could be used to evaluate different design changes aimed at reducing the amount of data communicated even further.

8.8.3 Explaining overall Penny performance

We ran various combinations of Penny—using eight versions of Penny itself (with different improvements and modifications), four benchmark inputs, and three levels of parallelism (4, 8, and 16 workers). We measured the median time out of 31 runs, giving us 128 timing points. For each point, we used SIMICS to generate 12 aggregate values, covering the different cache and TLB miss types in the target system.

We next selected three variables that we presumed to be important for explaining performance, namely the number of memory reads, number of read misses to the first level cache, and the number of misses to the second level cache. A multiple regression of these variables against the database, and fudging, results in:

$0.1 * Reads + 0.33 * ReadMisses_{L1} + 10 * ReadMisses_{L2}$

The correlation to the actual execution time is 0.96, not exceptional but clearly a good indicator. The performance of large configurations (16 processors) is overestimated and, conversely, the small configuration (4 processors) is underestimated. We suspect the reason for this is that we are lacking a fourth coefficient to measure bus contention, an issue that becomes significant as the number of communicating processors increase.

Misses to the second level cache cause the bus load to increase. The bus is a globally shared resource, so when it approaches saturation it stalls new accesses. As we increase parallelism, the miss rate of reads and writes both increase, which is natural since we are spreading work and communicating more. At the same time, execution time is decreasing, compounding pressure on the bus. This could explain the abnormally high coefficient for read misses to the second level cache, which of course is the one of the three closest correlated with bus contention and thus has to “carry” the bus contention load in the regression.

Instruction-level simulation is a powerful tool when a parallel system is constructed. The cache performance is so important for the overall performance of the system that it is hard to tune a system without having access to cache miss statistics.

A parallel system can have a perfect behavior with respect to the number of instructions executed and still not show linear speedup. Only when the cache misses are taken in to account can the performance of the system be predicted.

Chapter 9

An Evaluation

In this chapter I present an evaluation of the Penny system. The system shows good parallel performance even for programs with fine-grain tasks. A large real-life program, the compiler compiling itself, shows that automatic parallelization works but also reveals its shortcomings.

The cache performance is investigated using instruction-level simulation. The miss rate of the second level cache approaches 2% for parallel programs with intensive inter-process communication. For some programs this is the limiting factor.

The garbage collection time is for some benchmarks dramatically reduced, a factor of sixteen is observed using two workers. I show that the active data can be reduced as an effect of process interleaving. The reduced amount of active data reduces the garbage collection time. For other benchmarks the garbage collection time is a severe bottleneck. For non-deterministic computations the garbage collection is an inherent limitation.

9.1 The experimental setting

The experiments were done on a SPARCCenter-2000 with twenty processors. The load on the machine was normally low and there was no problem in obtaining reliable figures.

9.1.1 Binaries

To evaluate the system several versions of the Penny system have been used. All timings are done using a version compiled with an optimizing compiler (GNU gcc 2.7 -O4). The same version has been used for all benchmarks. The system is complete with garbage collection and environment identifiers on structures and list cells.

Two specially compiled versions have been used to collect statistics. One gathers information about binding depths, type of executed tasks and the depth on which tasks are stolen. This version should, for all but extreme cases, be close to the actual execution. The second creates a trace file where events are logged chronologically with a time-stamp. Since a lock is used to sequentialize the logging operation, the behavior of the system is changed. This version was only used for logging of scheduling events. These are infrequent and should not, except for execution time, alter the execution.

9.1.2 Timings

The execution time does not include the initialization of the system. The initialization includes memory allocation, loading of code, creation of Solaris threads, and initialization of the Penny workers. The clock is started when all workers are prepared to enter the scheduler. The benchmark is only executed once for each timing i.e. it is a cold start for the AKL program. The clock is stopped when the threads that implement the workers terminate. The execution times reported contain, if not explicitly stated, the garbage collection time. All timings are wall time, measured by the Solaris **gettimeofday()** system call.

Enough memory was allocated in the initialization to avoid allocating new blocks during runtime. The size of the heap is for each benchmark fixed i.e. the size is not dependent on the number of workers. The initial heap size is for most benchmarks set to 4000 blocks of 8Kbytes each (32 Mbytes). The garbage collection limit was usually set to 3600 blocks. Up to 800.000 free-list blocks (64bytes each) for reclaimable structures were allocated. Re-allocation of stacks was turned off to avoid interference from memory allocation system-calls.

9.1.3 The hardware

The SPARCCenter 2000 (SC2000) is a bus-based shared-memory multiprocessor from Sun Microsystems. Each processor has two on-chip caches, one for instructions and one for data. On the SC2000's processors, 50MHz SuperSPARCs, the data cache is 4Kbytes, one-way associative with 32 byte cache lines. The instruction cache is 20Kbytes, five-way associative with 64 byte cache lines.

Two processors were actually faster (16Kbytes four-way associative first level data cache) than the remaining eighteen. These processors are about 7% faster. It has not been possible to prevent the operating system from using these processors. This has caused some anomalies when running on one or two workers but it does not significantly influence the results.

We did not have exclusive control of the machines so the benchmarks were executed while other users where using the machine. This did not prevent us from obtaining reliable figures.

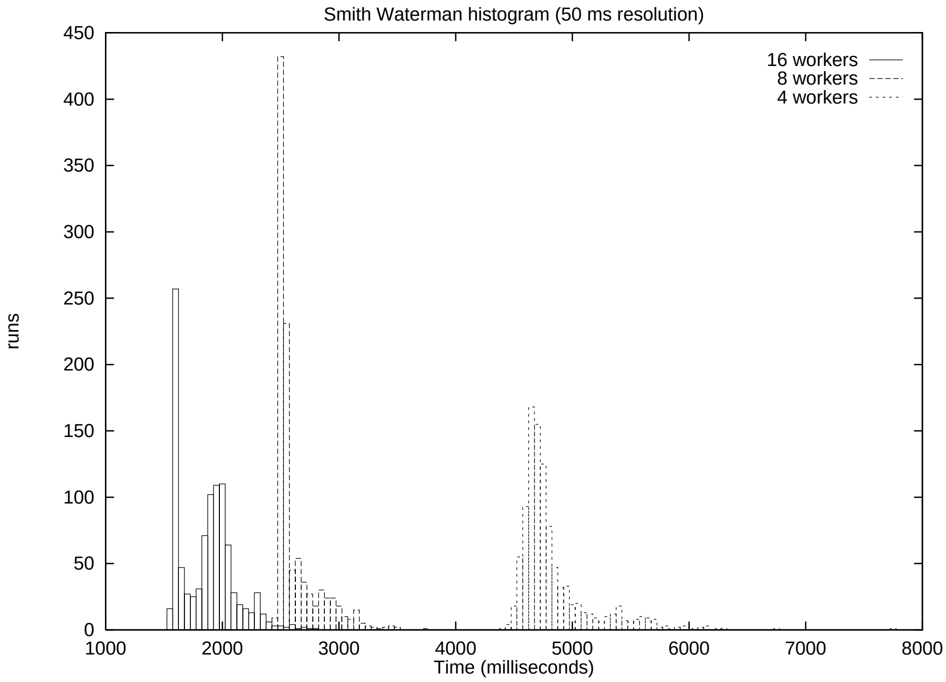

Figure 9.1: Histogram of execution time

9.1.4 Speedup

We measure parallel performance by executing the same benchmark on an increasing set of processors. This is the traditional way of presenting performance of a parallel system. We do not change the problem size with the number of workers. This is probably closer to how we will use increased processing power but the traditional method was chosen since it can be a problem to select benchmarks with different problem size.

The median execution time is used for comparison. The median execution time is chosen since it is less influenced by the few slow executions that are caused by operating system fluctuations. The median execution time gives the user better information about the typical performance of the system [31]. The difference between the median and mean is in reality not big.

Figure 9.1 illustrates what can be expected in a typical execution. The histogram shows the the distribution of execution time of the Smith-Waterman benchmark from three thousand runs using four, eight and sixteen processors. The distributions are as expected slightly skewed. The bimodal distribution that occurs when sixteen workers are used is a typical result of an additional garbage collection and memory allocation that were performed during runtime. In the benchmarks in the following sections, sufficient memory has been allocated at boot-time to avoid this. The four worker execution is slightly influenced by the heterogeneous hardware that we are using.

When showing execution time, we have chosen a log-log diagram of time vs. workers. A straight line means that the execution time, t, is a function of the number of workers, w, and two constants k and c.

$$t = k \times w^c$$

The constant k is the execution time using one worker. If the speedup is linear the constant c is -1. Speedup diagrams are used when the actual execution time is not essential.

9.2 Simple recursion

In the first test we have chosen benchmarks that should show good parallel performance. The benchmarks are simple to analyze and should have a predictable behavior. These benchmarks are a first test of the system. Good performance is a necessary but not sufficient condition for the validation of the system. The benchmarks are often used in the logic programming community.

matrix: A 500x500 matrix of floats multiplied by a vector. The execution sequentially spawns a task of fixed size in each recursion.

hanoi: The towers of Hanoi using 18 bricks. Each task spawns two tasks of equal size in each each recursion.

fib: The Fibonacci function calculating fib(27). Each task spawns two tasks of different size in each each recursion.

tak: The Takeushi function calculating tak(20,10,4). Each task spawns four tasks of different size in each each recursion.

The Fibonacci and Takeushi benchmarks are numerical benchmarks, so they do not create any data structures other than integers. The matrix benchmark creates lists to represent both the matrix and the vector. The hanoi benchmark creates a list of length 2^{18}.

9.2.1 Speedup

Figure 9.2 shows the minimum and maximum execution time of a hundred runs. As clearly seen they all behave well. The maximum execution time is less then 10% higher than the minimum execution time when eight workers are used. The difference is significantly larger (up to 27%) when fourteen workers are used but are even then remarkable stable. When more than eighteen workers are used almost anything can happen but the minimum execution time does still decrease. In the following sections only the execution time up to eighteen workers is reported.

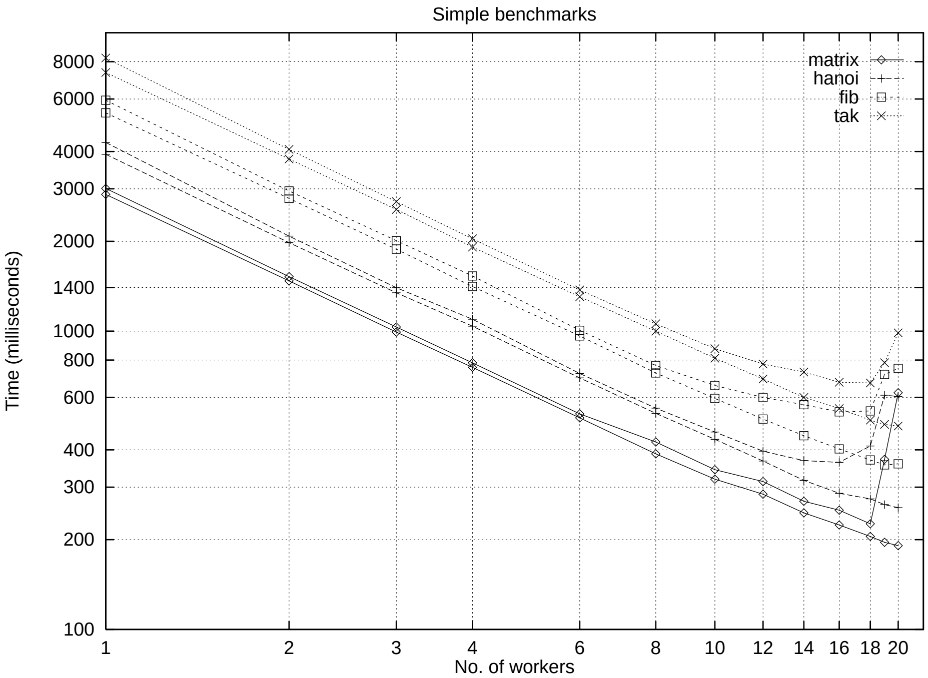

Figure 9.2: Simple recursive benchmarks

Workers	1	2	3	4	6	8	12	16	18	20
matrix	2933	1497	1017	772	519	397	288	231	213	197
hanoi	4190	2046	1386	1082	714	541	379	314	300	401
fib	5803	2915	1985	1506	986	750	541	446	429	424
tak	8037	3987	2658	2006	1344	1024	721	594	568	556

Table 9.1: Simple benchmarks, median execution time

The median execution time shown in Table 9.1 is close to the minimum execution time. One can notice that the minimum execution time using one worker and two workers is significantly better than the median. This is a result of the hardware that we used. When the faster processors were allocated the minimum execution time was improved. Otherwise the median execution time follows the minimum time until eighteen workers are used.

If we look at the speedup, based on the median runtime, shown in Table 9.2 we see that the speedup is almost linear up to eight workers. The system then performs slightly worse and only give a speedup of about 14 when eighteen workers are used. If the minimum execution times are compared the final speedup for twenty workers are for all benchmarks above 15.

The super-linear speedup obtained for the hanoi and matrix benchmarks is not a result of true speedup. It is a result of the fact that we are dividing medians. The

Workers	2	3	4	6	8	12	16	18	20
matrix	1.96	2.88	3.80	5.65	7.39	10.2	12.7	13.8	14.9
hanoi	2.05	3.02	3.87	5.87	7.74	11.1	13.3	14.0	10.4
fib	1.99	2.92	3.85	5.89	7.74	10.7	13.0	13.5	13.7
tak	2.02	3.02	4.01	5.98	7.85	11.1	13.5	14.1	14.5

Table 9.2: Simple benchmarks, speedup

Workers	1-2	2-4	3-6	4-8	6-12	8-16	10-20
matrix	1.96	1.94	1.96	1.94	1.80	1.72	1.63
hanoi	2.05	1.89	1.94	2.00	1.88	1.72	1.50
fib	1.99	1.94	2.01	2.01	1.82	1.68	1.50
tak	2.01	1.99	1.98	1.96	1.86	1.72	1.51

Table 9.3: Simple benchmarks, scaling

distributions are different depending on the number of workers and this affects the median.

Table 9.3 shows the achieved speedup when doubling[1] the workers. This is a nice way of illustrating how the performance decreases as the number of workers grows larger. It can clearly be seen that the speedup is almost perfect even when going from four to eight workers while going from six to twelve gives us a factor of 1.8 to 1.9. There is less use in doubling a ten processor system since this only gives a factor of 1.5 to 1.6 in performance.

These benchmarks are almost ideal as parallel benchmarks, they all divide up into many non-communicating processes. The creation of tasks is however different in each program. The binary spawn, in Hanoi, is of course the most profitable since it quickly divides the available work into equal size tasks. The scheduler does of course not guarantee that the initial allocation is perfect but any initial unbalanced allocation is rapidly balanced as workers run out of tasks. The linear spawning of tasks in the matrix program has a disadvantage that we investigate further in Section 9.3.

[1]The figures given for $10-20$ are the ratio of the execution time using ten workers and either nineteen or twenty workers since the median execution time using twenty workers sometimes is worse than for nineteen workers.

Workers	2	4	8	16
matrix	0.028 %	0.042 %	0.060 %	0.121 %
hanoi	0.007 %	0.007 %	0.013 %	0.024 %
fib	0.006 %	0.007 %	0.024 %	0.111 %
tak	0.006 %	0.009 %	0.030 %	0.210 %

Table 9.4: Simple, read cache performance

9.2.2 Cache performance

The read miss rate of the second level cache is important, as described in Section 8.8.3. A read miss is roughly 100 times as expensive as a hit in the first level cache. Even small miss rates of less than one 1% therefore has a significant effect on the total execution time.

Table 9.4 shows the read miss rates of the second level cache. The read miss rates are low for all benchmarks and do not severely limit the obtained speedup. These figures serve as a reference when cache performance is evaluated for the benchmarks in the following sections.

The matrix benchmark has a significantly higher miss rate, this could be due to the initial reading of the matrix. The miss rates for the Fibonacci and Takeushi benchmarks are initially low but then increase when sixteen workers are used. In both benchmark there are dependencies between the arithmetic operations. The dependencies cause suspensions which in turn cause read misses. The hanoi benchmark has the lowest miss rate. This benchmark is free of dependencies that could cause suspensions.

9.2.3 Load's up

What happens to the performance when the system is overloaded? Running benchmarks on a rebooted machine on a Sunday night is nice but does not answer the question of how the system behaves under normal conditions. To simulate a machine with high load we executed the benchmarks on an eight CPU machine instead of the twenty CPU machine normally used.

Figure 9.3 shows the median execution times of twenty runs. As clearly seen the system performs well up to eight workers. When more workers are used the performance degrades but does not become worse than the performance of four workers.

The system performs well since a busy worker can continue to work even if other workers are not scheduled by the operating system. An idle worker can, in the same way, steal work from a busy worker even if the busy worker is not scheduled for execution.

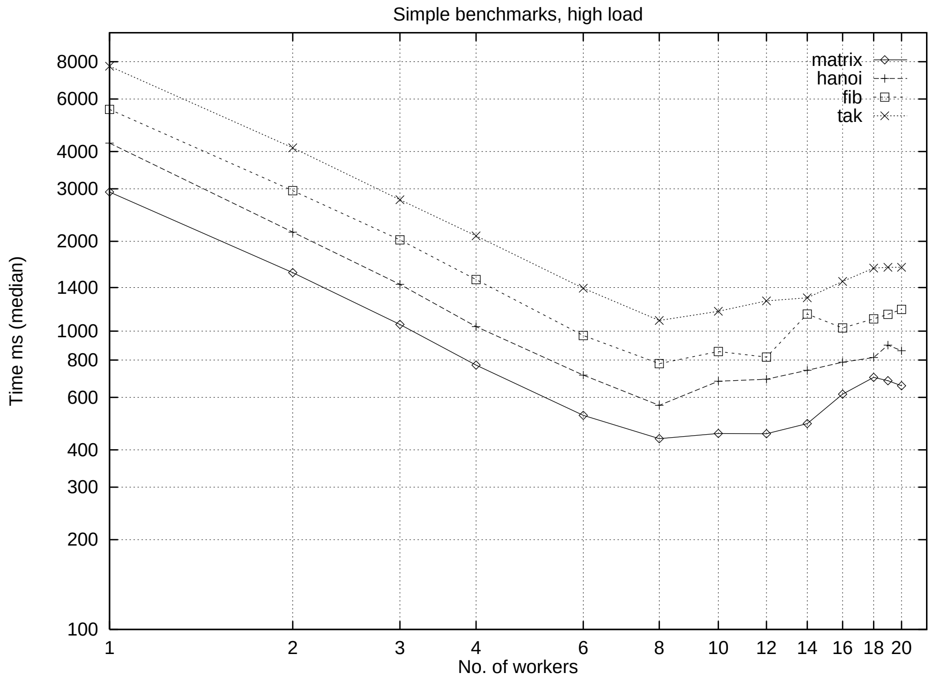

Figure 9.3: Simple benchmarks on a loaded machine

In an experiment with a scheduler, where an idle worker would have to rely on a busy workers to delegate the task, the system broke down when executing on a loaded machine. Any interactions between two workers where both workers have to be scheduled is dangerous. In the Penny system all operations in an interaction are performed by a single worker. The interaction is protected by a lock and only if the worker that holds the lock is preempted during the critical section will other workers suffer. The critical sections are small and this seems not to be a problem.

9.3 The limitations of sequential task creation

It is important to write programs in a way that there is no inherent limit on speedup. A sequential dependency between task creations, present in matrix multiplication as usually written, is such a limit. The limitation reduces the speedup as well as imposing a minimum execution time.

The matrix multiplication benchmark, partly listed in Figure 9.3, consist of three parts: the creation of a vector (filled with floating point numbers), the creation of the matrix and, the multiplication of the vector and the matrix. The multiplication process creates a continuation task in each call of `vmul/4`. The continuation task is the key to the parallelization of the process. The `vmul/4` procedure cannot be divided into subtasks since it is a recursive definition that consist only of unification and built-in calls. Notice however that the creation of the vector and matrix can be executed in parallel with the multiplication.

```
bench(N,M):-
      ->    makevector(N, Vector),
            makematrix(M, N, Matrix),
            multiply(Matrix, Vector, _Result).

multiply([],_,Res):-
      ->    Res = [].
multiply([V|Rest], Vector, Result):-
      ->    Result = [R|Others],
            vmul(Vector, V, 0, R),
            multiply(Rest, Vector, Others).
```

Figure 9.4: Matrix multiplication

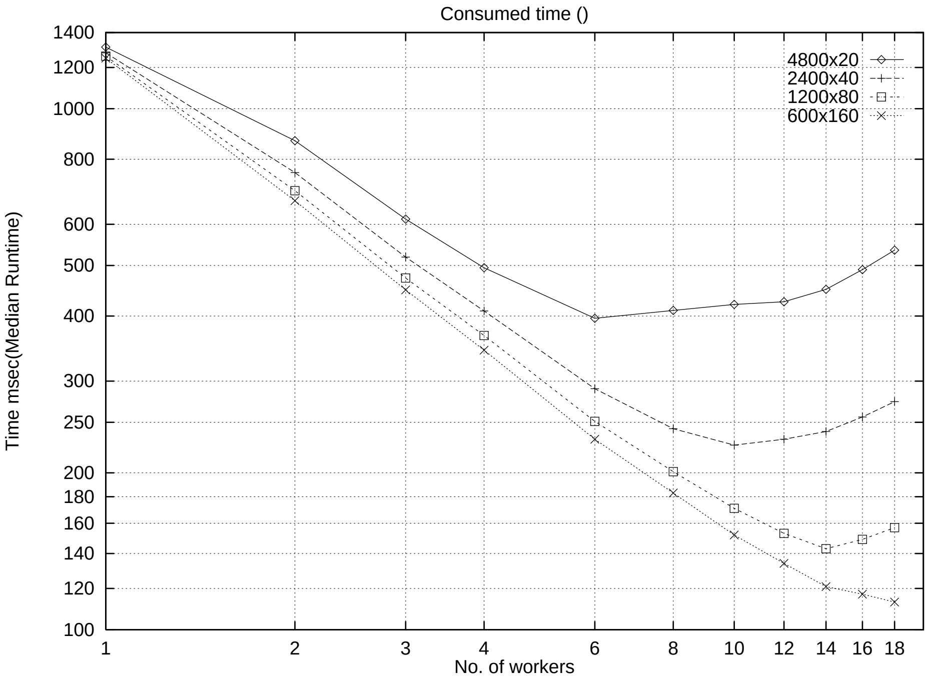

Figure 9.5: Matrix multiplication

Worker	1	2	3	4	6	8	10	12	14	16	18
4800x20	1313	868	614	495	396	410	421	426	450	491	535
2400x40	1280	754	519	409	290	243	226	232	240	256	274
1200x80	1261	696	473	367	251	201	171	153	143	149	157
600x160	1250	666	449	344	232	183	152	134	121	117	113

Table 9.5: Matrix multiplication, median (100 runs) execution time in milliseconds

To investigate matrix multiplication further, four benchmarks were executed. The benchmarks multiplied a matrix of size $M \times N$ with a vector of length N. The product $M \times N$ is equal for all benchmarks so the number of floating point operations is equal in all benchmarks. Each benchmark was executed a hundred times and the median execution time was selected. The benchmarks give an estimate on the smallest useful granularity. The smallest useful granularity is a function both of the scheduling overhead and the number of workers used.

The execution time is shown in Figure 9.5 and Table 9.5. There are four important observations that can be made:

- the execution time using one worker is almost identical for all benchmarks,
- the execution time using two workers is significantly different,
- the speedup from two to six workers is almost constant, and
- a sharp knee is reached when using 4, 6, and 14 workers.

The explanation to this is to find in number of scheduling operations and the size of the task.

9.3.1 The task size

We estimate the time, V, of a single `vmul/4` call, as:

$$V = (T_1 - C)/M$$

where T_1 is the total execution time using one worker, and C is the time to create the matrix and the vector. Table 9.6 shows the calculated time V given the median execution time T_1, and C.

The time to create the matrix and vector, C, is directly proportional to $M + 2N$, since the same vector is used for all rows, i.e only two vectors are actually created. The calculated time V is as expected directly proportional to N, the length of the rows.

Benchmark	M	T_1	C	V
4800x20	4800	1313	36	0.26
2400x40	2400	1280	19	0.52
1200x80	1200	1261	12	1.04
600x160	600	1250	8	2.06

Table 9.6: Time to do one vector multiplication in milliseconds

Workers	O_2	O_3	O_4	O_6
4800x20	212	177	167	-
2400x40	114	93	89	77
1200x80	66	53	52	41
600x160	41	33	32	24

Table 9.7: Matrix multiplication, overhead compared to ideal

9.3.2 Analysis of initial speedup

We first only consider the timings using up to six workers. There is a significant difference in the speedup between the benchmarks. Table 9.7 shows the total overhead O_W when more than one worker is used. The overhead is calculated as

$$O_W = T_W - (T_1/W)$$

where T_W is the median execution time using W workers.

The overhead is proportional to the number of rows in the benchmark. It can be modeled with the function

$$O_W = (M/W)S_W + B_W$$

where (M/W) is the number of rows executed by one worker and S_W and B_W are parameters to be determined. If we omit the six-worker figure for the first benchmark, linear regression gives the values in shown in Table 9.8 with correlation equal to 1.0. The six-worker figure for the first benchmark is influenced by other limitations that are explained in the following section.

We know that the structure of the matrix benchmark induces a scheduling operation for each of the M rows in the matrix. When one worker is used this is a local operation but when several workers are used a global scheduling operation is performed. The parameter S_W describes the overhead induced in each scheduling operation.

The total execution using two workers includes in the first benchmark 4800 scheduling operation. In the last benchmark only 160 scheduling operations are performed.

Workers	2	3	4	6
$\mathbf{S_W}$	0.081	0.10	0.13	0.18
$\mathbf{B_W}$	17	12	13	6

Table 9.8: Matrix multiplication, scheduling overhead

This explains why the total execution time using two workers is different for the benchmarks.

The calculated value of S_W increases as the number of workers increase. This is natural since searching for a task in the scheduler is more complicated as the number of workers increase. The equation

$$S_W = 0.024W + 0.032$$

correlates exactly (1.0) to the obtained values. The scheduling overhead when two workers are used is thus 0.08 milliseconds but increases with 0.024 milliseconds for each worker that is added.

If S_W was constant the speedup between two and six workers would for be identical for the benchmarks. Since S_W increases with W the benchmark with the most scheduling operations is penalized.

The total execution time is given by

$$\begin{aligned} T_W &= (T_1/W) + O_W \\ O_W &= (M/W)(0.024W + 0.032) + B_W \end{aligned}$$

where the value of B_W is found in Table 9.8.

Apart from the figure B_W we have a perfect description of the initial overhead for all of the benchmarks. The parameter B_W is complex to analyze since it is related to how the matrix is created, initial cost of running in parallel etc. The parameter constitute less than two percent of the execution time.

Even if we have successfully understood the difference in the initial speedup there is still one question that is unanswered. Why does the execution time level-out, and why does it level-out at different levels?

9.3.3 The minimum execution time

It is the scheduling overhead that explains the initial difference between the benchmarks but it does not explain the existence of the minimal execution time. To explain this we have to understand the scheduling operation in detail.

When a worker executes the `vmul/4` procedure call it also adds a continuation task on its own stack. This task is referring to the recursive `multiply/3` procedure call

Benchmark	W	T_W	D_W
4800x20	6	396	0.08
2400x40	10	226	0.09
1200x80	14	143	0.12

Table 9.9: Estimated turn-around time, D_W

and is the key to the rest of the multiplication. A worker that steals the task has to initiate the call to the vector multiplier for the next row before a new continuation task is produced. This continuation can then be stolen by another worker. Observe that there is a single continuation task in the whole system that is moving around.

The time duration between the two steal operations in the system is called the turn-around time D. The total execution time can never be lower than $M \times D$. If we divide the obtained minimal execution time (as shown in Figure 9.5) with the number of rows in the matrix, we get an estimate of the turn-around time for the given benchmarks shown in Table 9.9. The figure for the last benchmark is left out since we did not reach the limit of the system with the available twenty processors.

The turn-around time D, the time to multiply one vector V, and the scheduling overhead S_W sets a limit on how many workers that can productively take part in the execution. The limit is roughly $\lceil (V + S_W)/D \rceil$. When a worker starts to execute a task it works for V milliseconds, and remains idle in the scheduler for S_W milliseconds. During this time a scheduling operation can be performed every D milliseconds. If more workers are allocated they only wait for the continuation task to be passed around. This is why we see the sharp knees in the diagrams in Figure 9.5.

The smallest useful granularity therefore depends on the number of workers that take part in the execution. In a six worker execution the task size should not be smaller than 0.26 milliseconds. The more workers to keep busy, the higher the needed granularity.

Explaining the performance of a parallel system is hard even for a regular benchmark like matrix multiplication. One solution is to simply say that “things first get better and then get worse” but it is rewarding to track down the figures from a benchmark. In this example there are still things that are not explained. Although the performance is not a problem it is important that one can describe exactly the behavior of the system. If the performance does not agree with the calculations the understanding of the system is wrong or at least not complete.

```
multiply_t(v(V0), V1, R0, R):-
     ->   R0 = [V|R],
          vmul(V0, V1, V).
multiply_t(m(ML,MR), V, R0, R):-
     ->   multiply_t(ML, V, R0, R1),
          multiply_t(MR, V, R1, R).
```

Figure 9.6: The matrix represented as a tree

9.3.4 Remedy

The speedup limit due to sequential task creation can be circumvented by rewriting the program to create tasks in parallel. The system already allows tasks to be created and scheduled in parallel, if the tasks are stolen from different workers. The system cannot do this for the matrix program as written, but a small change to the program does the trick. If we represent the matrix as a tree instead of a list (as in Figure 9.3.4), the system has no difficulties in obtaining good speedup even for matrices with small rows. Other techniques to transform sequential programs are described in [18, 28].

9.4 Communicating processes

The benchmarks in the previous section are all programs that divided up into more or less independent parts. The speedup is as expected good, the only problem is task granularity. What happens when the AKL processes depend on each other? In the following benchmarks I try to evaluate the consequences of process communication.

9.4.1 Stream and-parallelism

These programs divide up into processes that communicate through streams of data. The number of processes can be fixed or be a function of the input. The programs can often be executed "from left to right" without any suspensions.

The programs that we have chosen are all well-known benchmarks in the committed choice community [67]. The benchmarks are:

mastermind: The mastermind puzzle by E. Tick, using two guesses and three colors.

kkqueen: The candidates/non-candidates queens program by K. Kumon and E. Tick, using 9 queens.

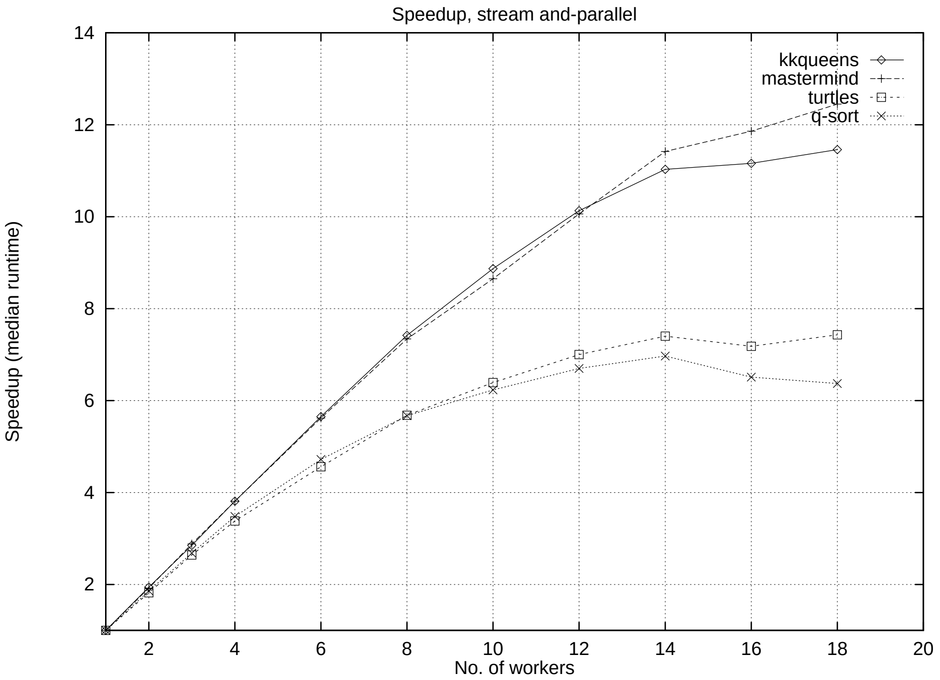

Figure 9.7: Stream parallel benchmarks

turtles: The turtles puzzle by E. Tick using layered streams.

qsort: Quick-sort of the first 10.000 four-digit numbers extracted from the decimals of π.

The quick-sort program is not always seen as a stream-parallel program since the parallelism that is exploited is often limited to "first divide the list, then sort the two parts in parallel". In a fine-grain parallel system the sorting of the two sub-lists can be done in parallel with the partitioning of the list thus increasing the amount of parallelism.

The median execution time, for sets of one hundred runs, is shown in Table 9.10. As shown in Figure 9.7 the system behaves well for mastermind and kkqueen but levels off for turtles and qsort. In the qsort program any execution must traverse the initial list once and this can only be done sequentially. This sets a lower limit on the execution time regardless of the number of workers. Since the quick-sort algorithm sorts a list of length n in $n\lg(n)$ steps and even the best parallel execution requires n steps, the optimal speedup is $n\lg(n)/n$ i.e. $\lg(n)$. For the quick-sort benchmark this would mean an ideal speedup of lg(10000) i.e. 13.2 assuming a perfectly balanced execution tree.

Since the actual maximum speedup is only 7, there is another limiting factor in this benchmark. The turtles program has no obvious algorithmic limitation and still the speedup is only 7.4. We run the benchmarks using SimICS to get the statistics listed in Table 9.11. The read cache performance is clearly worse for the quick-sort and turtles benchmarks. When eight or sixteen workers are used the miss rate is almost a magnitude higher than for the mastermind benchmark. These figures should also

Workers	1	2	3	4	6	8	10	12	14	16	18
mastermind	3037	1578	1053	797	540	414	351	302	266	256	244
kkqueen	3851	1984	1351	1011	681	519	434	380	349	345	336
turtles	1924	1055	730	570	422	339	301	275	260	268	259
qsort	2306	1241	862	663	489	407	370	344	331	354	362

Table 9.10: Execution time, stream and-parallel benchmarks

Workers	2	4	8	16
mastermind	0.021 %	0.029 %	0.057 %	0.15 %
kkqueen	0.017 %	0.019 %	0.030 %	0.11 %
turtles	0.11 %	0.19 %	0.37 %	0.65 %
qsort	0.051 %	0.20 %	0.46 %	0.97 %

Table 9.11: Stream parallel, read cache performance

be compared to the figures in Table 9.4. The miss rate is significant and severely limits the obtainable speedup.

9.4.2 The game of Life

This program is interesting since it makes use of concurrency not only as a means for parallel execution but as a programming tool. The program is extreme in the ratio of communication over calculation. This means that the dominating factor of the execution time does not come from instruction decoding but rather from the suspension mechanism. This program pushed the system to its limits when it comes to waking suspended goals.

The game of Life is implemented as a two dimensional grid of communicating processes. Each cell in the grid is represented by an AKL process. The grid has the shape of a torus. The program consists of two parts: the building of the toroid and the communication between cells. The building phase only takes about 5% of the execution time and shows in itself good speedup. The building phase is of course executed in parallel with the computation of the first generation. Each cell is implemented by the `cell/12` process listed in figure 9.8. The first argument is the number of generations, the second and third the current and final state, the fourth the stream of the states. The remaining arguments are the streams from the neighbors.

The `call/12` process suspends, waiting for information from all of its neighbors, and then calculates its own next state. When a cell process is executed, in average, four

```
cell(0, State, Last, This, _, _, _, _, _, _, _, _):-
    ->  Last = State,
        This = [].
cell(G, State, Last, This,
    [NW|NWi], [N|Ni], [NE|NEi],
    [W |Wi], [E |Ei],
    [SW|SWi], [S|Si],[SE|SEi]) :-
    ->  dec(G,G1),
        T is NW + N + NE + W + E + SW + S + SE,
        change(State, T, New),
        This = [New|Rest],
        cell(G1, New, Last, Rest, NWi, Ni, NEi, Wi, Ei, SWi, Si, SEi).
```

Figure 9.8: The cell of life

Workers	1	2	4	8	16
10 30x30	2617	1377	843	561	460
10 60x60	10543	5366	3257	2192	1826
10 120x120	42366	21469	12860	8697	7283
40 30x30	10195	5374	2981	1734	1258
160 30x30	40279	21367	11394	6553	4476

Table 9.12: Life, execution time in milliseconds, excluding gc

of the eight neighbors already have calculated their new state. The cell therefore only suspends on four of its neighbors.

Speedup

The smallest benchmark uses a toroid of size 30×30 and computes 10 generations. In the larger benchmarks either the toroid or the number of generations is increased. Garbage collection was only needed in the two largest benchmarks and only took one to two percent of the total execution time. The median execution time of twenty runs, excluding garbage collection, is shown in Table 9.12. The execution time using one worker is directly proportional to the size of the board and the number of generations.

Figure 9.9 shows the obtained speedup based on the the median execution time, excluding garbage collection. The speedup is strongly dependent on the number of generations.

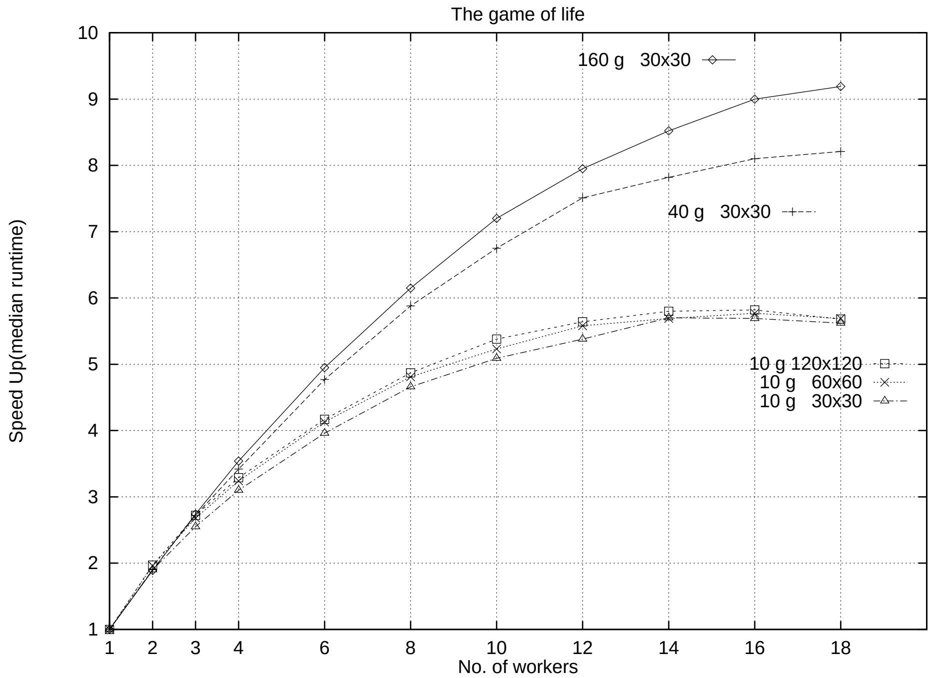

Figure 9.9: The game of life

Wake tasks

The reason for the difference in performance is to be found in the ratio between stolen and executed wake tasks. The wake tasks are the only important tasks to look at. There are no recall tasks in the program and few (fewer than 50) continuation tasks are stolen.

Let's first look at the "10-30x30" benchmark. There are $30 \times 30 = 900$ cells and each cell is computed 10 times. This means that there are $900 * 10 = 9000$ cell operations. When one worker is used each cell is suspended on four of its neighbors which means that $900 * 4 * 10 = 36000$ wake tasks have to be handled. The task are however not identical. Three out of four tasks only remove a constraint whereas the fourth promotes the and-box, suspends the goals of the next generation and wakes other suspended goals. As the number of workers increases the number of suspensions decreases only marginally. The overhead of stealing a task is, as shown in the matrix benchmark, high compared to a wake task that does not lead to a promotion.

If the worker is lucky it is able to continue the execution with locally generated wake tasks. If it is unlucky the generated tasks are stolen by other idle workers. As more workers are added the ratio of executed tasks over stolen tasks decreases. When sixteen workers are used, many of the task generated are stolen by others.

Table 9.13 shows the number of stolen and executed tasks in one run of each of the benchmarks. As expected the ratio decreases as more workers are used. The ratio increases as more generations are executed but decreases as the toroid gets bigger. Exactly why this is happens is hard to explain intuitively; it is probably easier to

Workers	1	2	4	8	16
10 30x30					
executed tasks:	36000	35999	35941	35783	35531
stolen tasks:	0	2372	4731	7982	9344
ratio:	-	15	7.6	4.5	3.8
10 60x60					
executed tasks:	144000	144000	143867	143147	142688
stolen task:	0	8753	21312	34240	40873
ratio:	-	16	6.8	4.2	3.5
10 120x120					
executed tasks	576000	575992	575549	572846	569059
stolen tasks	0	33603	87890	141089	193369
ratio:	-	17	6.5	4.1	2.9
40 30x30					
executed tasks:	144000	143994	143866	143473	142881
stolen tasks:	0	2494	6238	13524	16899
ratio:	-	58	23	10	8.5
160 30x30					
executed tasks:	576000	575977	575742	574267	571791
stolen tasks:	0	2485	10851	35838	52399
ratio:	-	232	53	16	11

Table 9.13: Stolen vs. executed tasks

find tasks to execute in the next generations of the woken cells than moving in the grid.

Cache performance

Running the benchmarks in SIMICS gives interesting figures on the read miss rate shown in Table 9.14. The miss rate increases as expected when the number of workers increase. The final miss rate of the "30x30" benchmarks is high. The figures are higher than for the quick-sort benchmark and more than twice the miss rate of the turtles benchmark.

It is also interesting to notice that the miss rate is better for the benchmarks with larger grid. This indicates that the workers spread in the grid and the larger grid re-

Workers	2	4	8	16
10 30x30	0.20 %	0.53 %	0.93 %	1.59 %
10 60x60	0.14 %	0.32 %	0.47 %	0.91 %
10 120x120	0.25 %	0.18 %	0.42 %	0.65 %
40 30x30	0.33 %	0.59 %	1.02 %	1.36 %
160 30x30	0.41 %	0.66 %	0.99 %	1.42 %

Table 9.14: Life, read cache performance

duces the conflicts. Notice that the miss rate actually decreases in the "10 120x120" benchmark when going from two to four workers. This could be a consequence of having more cache memory in total when running on four processors.

How significant is the cache performance? The total number of read operations in the "160 30x30" benchmark when using two workers is about 225 million. When sixteen workers are used the total is also 225 million; the extra number of workers do not increase the number of read operations. The nonlinear speedup could of course be explained by an imbalance in the distribution of work, but apart from the first worker that performs some extra work during garbage collection, the workers all perform 14 million read operations and differ only by 25 thousand operations. The first worker performs 14.9 million operations, only 6% more operations than the worker with the lowest read count. The work is thus balanced and there is no overhead in the number of operations that can explain the nonlinear speedup.

In Section 8.8.3 we found good correlation between the execution time and the number of read operations. The following equation was shown to have a 0.96 correlation to the obtained execution time:

$$0.1 * \text{Reads} + 0.33 * \text{ReadMisses}_{L1} + 10 * \text{ReadMisses}_{L2}$$

We know that about 10% of all read operations miss the first level cache. If we insert the figures for the "160 30x30" benchmark using sixteen workers, then we get the figure 3650 milliseconds. The actual execution time is 3439 milliseconds, including garbage collection. If we assume a second level read miss rate of 0.15%, which is the miss rate when running on one worker, then we get an estimated execution time of 1888 milliseconds i.e. a speedup of thirteen instead of seven. The cache miss rate is therefore a limiting factor for the "30x30" benchmarks.

In an experiment with a "160 120x120" benchmark the ratio of executed to stolen tasks was 23. The speedup, excluding garbage collection, was 12.5 when sixteen workers were used.

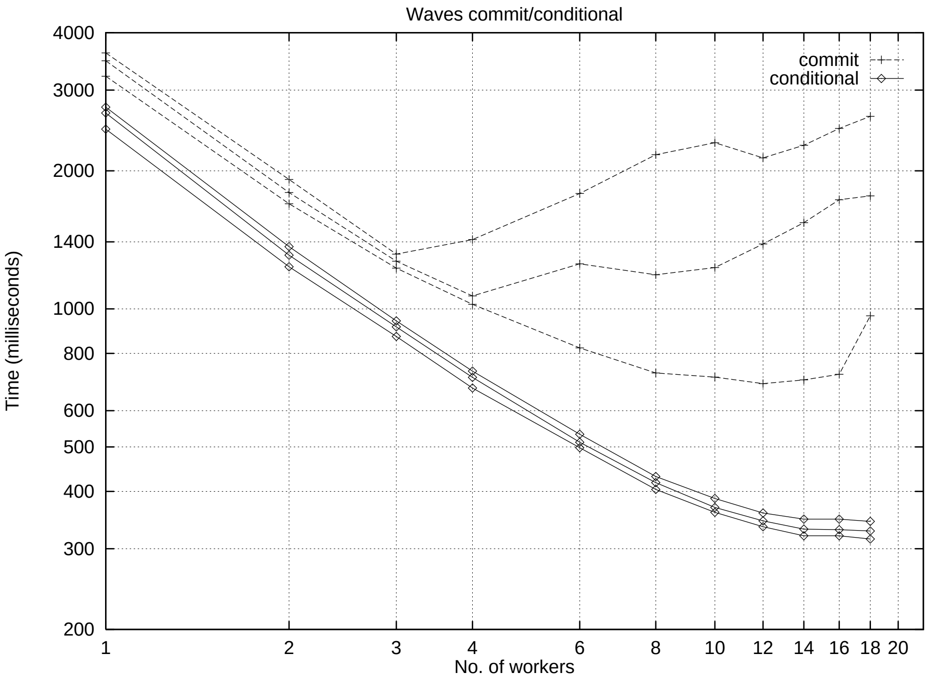

Figure 9.10: The importance of being lazy

9.4.3 Waves

This program was originally written by Ian Foster and later translated to KL1 by Evan Tick. The program build a torus and computes the sum at each point using an iterative technique. The benchmark uses a two dimensional toroid of size 9×9 and computes 5 generations.

A process is started at each point in the torus. The process is defined using seven clauses each with a complex arithmetic guard. Up to six arithmetic comparisons have to be performed in a guard to solve it. Since the program was translated from a KL1 program all the clauses were defined using the commit operator. It was only after a careful examination of each guard that I could convince myself of that the guards were actually mutually exclusive and that no guard could succeed unless the previous guards had failed. This made it possible to change the commit operator into a conditional operator which improved the performance.

The minimum, median and maximum execution times of one hundred runs for the two versions are shown in Figure 9.10. It is clear that there is a big difference between the two benchmarks. The reason for this is the change in guard operator.

When a conditional guard is used the system can stop the evaluation of guards if the first guard suspends. This is not the case when a commit guard is used. When the commit operator is used all the guards of a process have to be evaluated in order to find out if any guard succeeds. When only a few workers are used this results in an increased execution time but the system is still stable.

When more workers are used the workers tend to start the execution of process ahead of time. Since not enough information is available the processes suspend. It is much more expensive to wake a suspended guard than it is to execute it. The extra workers not only do unnecessary work ahead of time but they aggravate the situation.

An optimizing compiler does not easily solve the problem. The compiler can easily determine that certain variables have to be bound before any of the guards can succeed but the system would still have to execute the guards as soon as one variable is bound to determine if all guards fail. Only if the process is annotated to be a non-failing process can the system be lazy.

The problem would not occur in a committed choice language where failure of guards is something that can be ignored.

9.5 Non-deterministic programs

One of the benefits of AKL is that one can implement both and- and or-parallelism. The Penny system can execute several guards in parallel even in a deterministic program but this is not exploited eagerly. Only if several and-nodes under a choice-node are suspended and then woken is or-parallelism exploited in a deterministic program.

In non-deterministic programs, workers are invited to explore alternative solutions. When choice splitting is performed a task is created that refers to the right branch. This task can, as any other task, be stolen by another worker. The task is disguised as a wake task so the scheduler is not aware of the difference between deterministic and non-deterministic executions.

Parallel execution in a non-deterministic program is either simple or impossible. The simple solution is to look for all solutions to a problem. This is profitable to parallelize since the execution divides up into independent, often large, tasks. The impossible (or at least much harder) situation is when one wants to find the leftmost solution in a search tree. If the solution is at the far left in the search tree two workers do not find the solution faster than one worker. The second worker only does *speculative work*. In a system that is targeted to or-parallel execution the ability to avoid doing speculative work is crucial [4].

9.5.1 Speculative work

A simple queens benchmark can be used to show how unproductive speculative work can be. The program finds one solution in a sixteen-queens puzzle using a short circuiting technique. It is a non-deterministic program that uses a set of processes to determine if a set of queens threaten each other or if a deterministic choice of a queens position can be made. If no deterministic choice is found the position of a

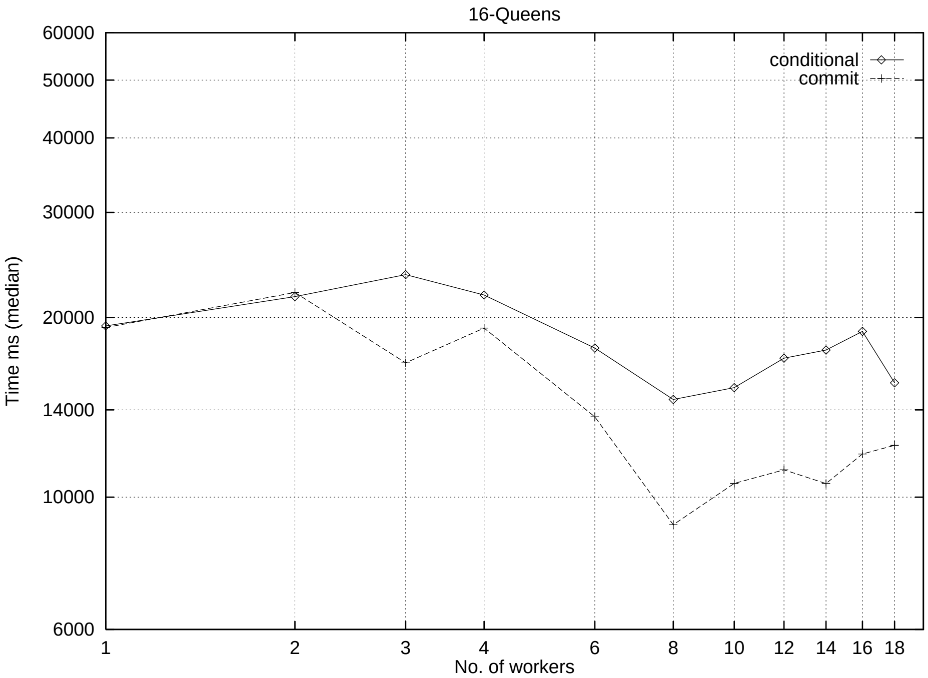

Figure 9.11: Speculative work

queen is guessed. The Penny system can utilize both the and-parallelism available through processes in an and-box and or-parallelism generated when a position of a queen is guessed.

Figure 9.11 shows the execution time (median of 40 runs) for two versions of the program, one that uses a commit guard operator and one that uses a conditional guard operator when the solution is selected.

The difference in the two programs is the context in which the non-deterministic program is executed. When only one worker is used the guard operator does not make any difference but when more workers are used the commit operator allows any solution to be promoted while the conditional guard operator only promotes the leftmost solution.

Using the conditional operator all work that is performed to the right of the leftmost solution is not only speculative but pointless. Since the search tree is big compared to the number of workers and the leftmost solution is found in the left part of the tree, few workers are involved in the execution of profitable branches.

The benchmark with the commit operator can, depending on how the workers are scheduled, find a solution much faster than if the tree is searched left to right. The execution time is however unstable. When eight worker are used the execution time varies between nine and seventeen seconds. The increased performance can be called speedup but is closer to good luck.

It is of course possible to write a small benchmark with the solution far to the right in the search tree. Such benchmark would of course show good speedup since there is little, if any, speculative work. This would however prove nothing (even if the diagram in Figure 9.11 would make a better impression).

These two benchmarks exemplify that speculative or-parallelism is not practical if there is not a good scheme to reduce the amount of speculative work.

9.5.2 All solutions

A more predictable way of exploiting or-parallelism is when all solutions in a non-deterministic computation are collected. This is done in an aggregate construct such as the **bagof/2** or **numberof/2** constructs. Three simple benchmarks were used to evaluate the performance of the aggregate constructs.

queens An implementation of the queens puzzle (10 queens) using a short circuiting technique. Each row and column on the board is supervised by a set of processes. If a queen is placed on a square the remaining squares are invalidated. If there is only one valid square left in the line, a queen is placed on this square. Similarly each diagonal is supervised by a set of processes. The benchmark needs 5904 split operations.

scanner The programs finds a pattern in a grid where the number of filled squares in each row, column and diagonal is known. Uses a more general version of the short circuiting technique that uses ports to communicate between processes. The reduced search space is essential to find a solution. The benchmark needs 2426 split operations.

f-queens A simple but fast Prolog like implementation of the queens puzzle (10 queens). Does not make use of concurrency to minimize the search space. Needs 35538 split operations but the copied state is small.

Figure 9.12 shows the execution time (median of 40 runs) for the three benchmarks. As clearly seen these benchmark are stable in their performance. They all behave good up until ten workers are used. The queens and scanner benchmark reaches a speedup of about seven while the f-queens benchmark reaches a speedup of almost nine.

The leveling off of the speedup is an indication of a sequential thread in the implementation. I have traced this to the function that searches for a candidate. When a worker enters the guard handler it takes the lock of both the parent choice-node and the current and-node. Only when a candidate has been found, and a template of a copy of the current and-node has been inserted, is the lock of the the choice-node released. As long as the lock is held other workers are prevented from entering the guard handler when executing in sibling and-nodes. The searching, and the creation of a template and-node, under a given choice-node is thus sequentialized.

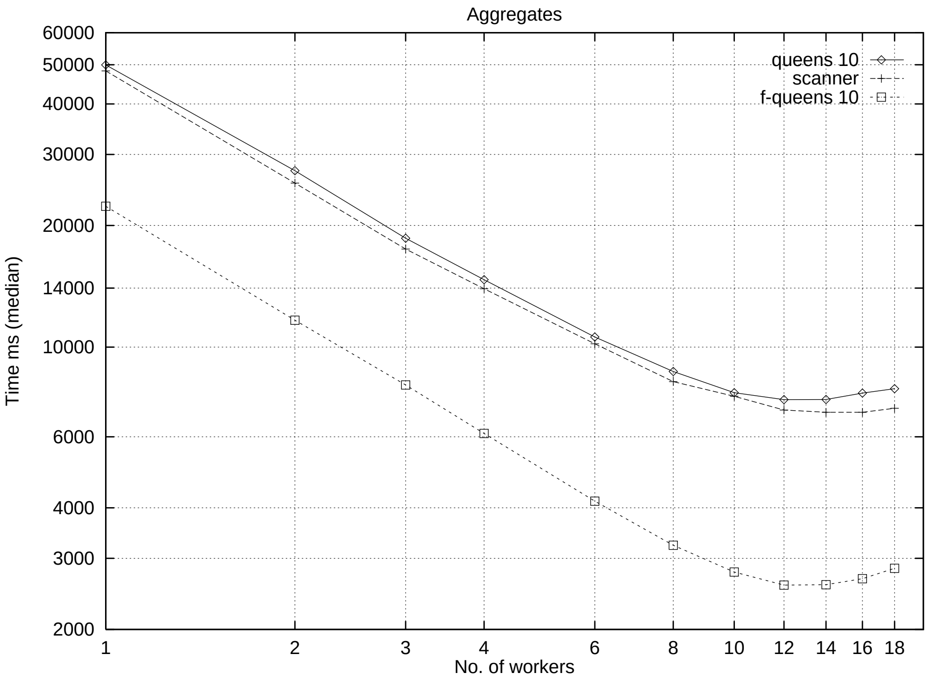

Figure 9.12: Aggregates collection all solutions

9.5.3 Garbage collection

The performance of the garbage collector is also a problem in these benchmarks. The garbage collection time (that was included in the previous figures) is shown in Figure 9.13. The garbage collection time increases in all of the benchmarks. For the scanner benchmark this is a severe limitation. The garbage collection time increases from about three hundred milliseconds to twelve hundred milliseconds. Compared to the total execution time this is an increment from less than a percent to seventeen percent.

When garbage collection is performed the workers traverse the whole execution state. The cache performance of this operation is poor. The scanner benchmark has a read miss rate of 6.5% when eight workers are used. The miss rate of the first worker, that is responsible for dividing up the execution state, is as high as 11%.

The garbage collector is however doing a good job. The reason for the decrease in performance is partly that the execution state increases when more workers are added. Each worker has about the same amount of living nodes to copy and is only penalized by having to synchronize its operations with other workers.

A more practical scaling in the benchmarks would of course be to let the available memory increase as the number of workers increase. This would not alter the amount of living data but it would make garbage collection less frequent.

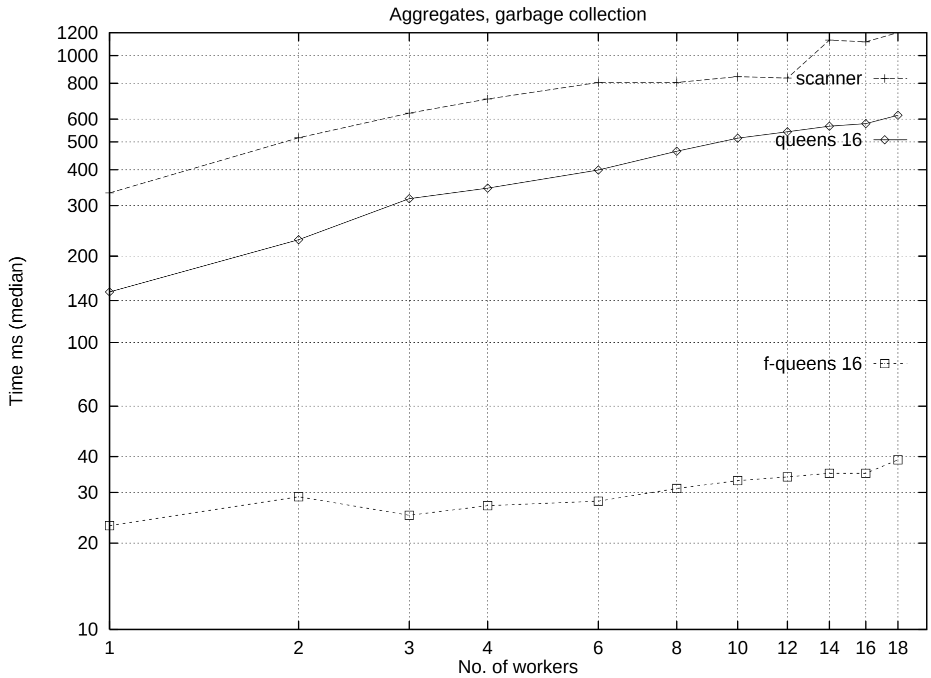

Figure 9.13: Garbage collection for aggregates

9.5.4 Nondeterminism in a reactive system

An interesting usage of parallelism is the ability to perform several non-deterministic computations in parallel. This is a more realistic setting, few programs consist of one large non-deterministic computation. An application could for example be a compiler where the register allocation is implemented by a non-deterministic computation. To simulate this a sequence of the benchmarks used in the previous section were executed in parallel. The execution time (median of 40 runs) is shown in 9.14.

Notice that the speedup is not as good as one would expect. The speedup is hampered by the fact that the workers are more eager to help each other with the guarded computations instead of starting new computations. This is a consequence of preferring a wake task over a continuation task. The performance of the system therefore suffers form the same drawbacks that were explained in section 9.5.1.

9.5.5 Deep computations

A Penny program can of course be arbitrary deep. The benchmarks used so far have all been using two or three levels. The first level is the main and-box. The third level has only been used by the non-deterministic computations that have encapsulated a search. This is however not the only reason for working with deep guards. The compiler, for example, uses five levels, not for encapsulating a search but because it is easier to write a predicate that succeeds or fails than writing a function that returns a truth value.

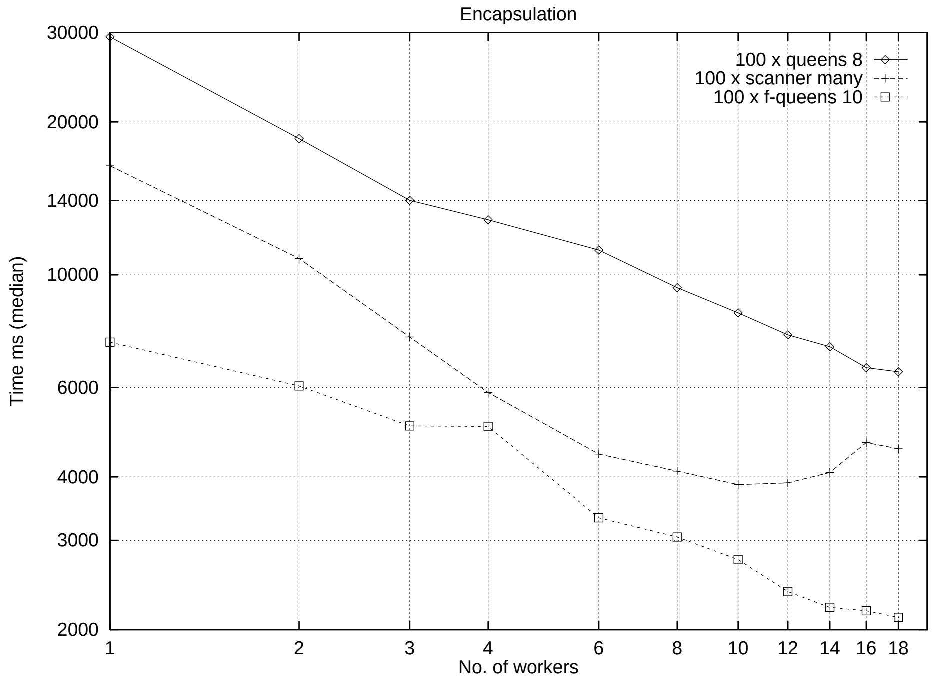

Figure 9.14: Non-determinism in a reactive system

Workers	2	3	4	6	8	10	12	14	16	18
1 (main)	9	30	42	76	110	134	169	203	255	333
2	143	648	303	848	1555	2201	3289	3714	6501	6557
3	5	65	29	266	168	445	737	1179	1494	2025
4	–	8	4	46	49	65	133	165	288	308
5	–	–	–	–	15	5	8	9	34	35

Table 9.15: Stolen tasks at different levels

The knights program is a non-deterministic program that uses several levels. It is an implementation of the "touring knight" i.e. place a knight in one corner of a board and jump to all squares without touching any square twice. The benchmark searches for all solutions (304) on a five times five board. This benchmark behaves well and a speedup of twelve is reached using eighteen workers.

Table 9.15 show the number of tasks stolen by workers on the different levels in the execution state. The majority of all tasks are stolen at level 2, this is the level where the different solutions are collected, but a substantial amount of tasks are collected at the third level. The number of tasks stolen at the fourth and fifth level shows that there is parallelism to exploit at all levels.

Scheduling of deep tasks adds to the complexity of the scheduler. A system where only tasks are distributed at the main level would be simpler to implement but would also limit the obtainable speedup. We have chosen to implement a scheduler

Workers	1	2	4	6	8	16	18
Larger picture							
total execution time (s):	2937	1356	737	494	370	196	179
garbage collection time (s):	366	23	24	25	25	29	41
Smaller picture							
total execution time (s):	27.5	15.0	9.03	7.66	6.91	5.84	5.05
garbage collection time (s):	0.091	0.015	0.365	0.454	1.74	1.73	1.46

Table 9.16: The ray tracer on two examples

that handles deep guards since the programmer otherwise would have to be aware of the limitations in the scheduler to get good performance.

9.6 The real world

The following evaluation is a test on how the Penny system performs when running larger programs or when running programs that were not intended as parallel benchmarks.

9.6.1 Ray tracing

A ray tracer program written by Johan Bevemyr in Prolog was easily converted to AKL and did after minor changes run well in parallel. The figures in Table 9.16 show the obtained execution times for two pictures. Notice that the execution time is reported in seconds i.e. the one worker execution takes almost 50 minutes.

To our surprise the figures showed super linear speedup. After celebrating for three days we started to investigate the figures further. The answer to the speedup was found in the reduced garbage collection time. The garbage collection time was reduced from 366 s using one worker to 23 s using two workers. This is even with a parallel garbage collection an exceptional speedup. What is happening?

The ray tracer program consist of three parts: 1) read the specification from a file, 2) calculate the picture and 3) write the result to a file. When one worker is used these three steps are executed one after the other. The representation of the generated picture grows as more rays are traced. In each garbage collection (192 collections) this representation has to be copied, this takes time. When two or more workers are used the three steps are executed in parallel. As the picture is calculated it is also written to the file. This means that the living part of the representation is small in each garbage collection.

The smaller example does not show the remarkable speedup. The picture is small and only four garbage collections are needed.

The garbage collector

It is noteworthy that the parallel garbage collector does not help us much in this program after the initial speedup. The garbage collection time in the larger benchmark increases after the first reduction from 23 seconds to 41 seconds or, as a ratio of the total execution time, from 3% to 23%. This highlights a deficiency in the design of the system that we have no simple solution to.

The parallel garbage collector divide the living nodes among workers. When a worker runs out of nodes, it tries to steal a node from another worker. If there are no nodes to steal, it looks for a sub-term to steal.

The problem in this program is that the dominating data structure is one large list of integers that represents the input specification. The worker responsible for the garbage collection of a structure copies a template of the structure and then start to copy the sub-terms of the structure starting from the end. In the list case this means that a new cons cell is allocated and that the worker continues with the cdr part. When a certain depth is reached the worker makes the list of un-copied car cells available for another worker to steal.

The work of filling in the car cells is small when compared to allocating the new cons cell in the first phase. The garbage collection time is bound by the time it takes to copy the list. A worker stealing the beginning of a list does not have much to do. In fact as indicated by the garbage collection figures these workers do more harm than good.

There are two orthogonal fixes to the problem that we have been thinking of. The first one is to have a generational garbage collector. This would improve the situation dramatically. In this program the whole input specification would be in the older generation after the first garbage collection and it could stay there during the execution. This is however not a complete solution to the problem since the garbage collection time still is bound to the copying of the longest list.

Another fix would be to divide the list up into chunks before the actual copying takes place. A worker that is given a list would then run as fast as it could n elements down the list. It would then divide the list making the first part available for another worker to steal. This would limit the garbage collection time to the time it takes to run down the longest list or the time it takes to copy a list of length n.

Disclaimer

Since this might give the impression that you can do ray tracing with exceptional speed using Penny, I must confess that the system is slow. Floating point arithmetic is so slow in the system that the execution time is a factor 100 from a C implementation. Nevertheless the program is only a factor three slower than an equivalent Prolog program executed by SICStus v.3 (emulated).

9.6.2 Smith-Waterman

The Smith Waterman algorithm computes a value that measures how good two sequences are aligned. The measurement is used when DNA sequences are compared with each other. The two sequences S of length m and T of length n are compared by evaluating a value $f_{ST}(n, m)$. The function $f_{ST} : N \times N \rightarrow D$ is defined as

$$f(i,j) = \begin{cases} g(f_{ST}(i-1,j), f_{ST}(i-1,i-1), f_{ST}(i,j-1) & \text{if } i > 0 \text{ and } j > 0 \\ \text{zero} & \text{if } i = 0 \text{ or } j = 0 \end{cases}$$

where *zero* is an element in D and $g : D \times D \times D \rightarrow D$ is a simple arithmetic function. The elements of the domain D are represented by a structure holding four integers.

This can be visualized as calculating the value in the lower right corner of a matrix where each element depends on its west, north-west and north neighbors. The execution obviously has to begin in the upper left corner and can then either compute the values one row, column or diagonal at a time.

Parallel execution

The program was written in Prolog by Margus Veanes and only small changes were needed to make it run in parallel in the Penny system. The original problem was to match one sequence of size 32 to each of one hundred sequences of equal size and find the best. The parallelism was obvious since the matching operations are independent.

An interesting challenge came from Roland Karlsson who argued that the obtained parallelism was not so interesting since the tasks are independent [36]. The real quest would be to make the algorithm itself run in parallel. It turned out that it was already running in parallel and the program only needed minor changes to give good speedup.

The biggest problem when tuning the program for parallel execution was to insert guards in order to delay the execution until sufficient data had arrived. The guards in a definition must not only select the right clause but also control the execution. If the guards are too weak the execution produces many suspended goals that later have to be woken. This is much more expensive than left to right execution without suspensions. If the guards are too strong the time to solve the guards is a problem. The goal is to insert the minimal guards that properly delay the execution.

Workers	1	2	4	6	8	12	16	18
256/32	3586	1846	939	641	496	388	357	356
64/64	3361	1718	876	596	459	332	276	259
16/128	3258	1666	853	598	456	347	286	272
4/256	3206	1648	863	632	507	375	318	298
1/512	3215	1785	945	670	534	416	370	358

Table 9.17: Smith-Waterman, execution time in milliseconds

In an early version of the benchmark the speedup was good but there seemed to be a limit to how fast the program could execute. The reason was found in the generation of the database. The sequences are generated using a random function and the passing of a seed to the random number generator made a perfect sequential thread that could not be parallelized. The procedure was then changed to allow the database to be generated in parallel.

Generic objects

To mimic the effect of an optimized Penny system, the arithmetic operation g that calculates the value of a position was implemented as a built-in. In order to do this a new data type was needed. This was easily done with the generic object interface. The new data type holds four integers and is used to represent the elements of the domain D. Only four built-in operations were defined: a recognizer, a constructor, a selector and the function g. The resulting program is about six times faster than the pure Penny program – faster than SICStus v3 native code.

Table 9.17 shows the execution time of the program for different data sets N/M where N is the number of sequences in the database and M the length of the sequences. Since the algorithm is quadratic we have reduced the number of sequences by four as the length was doubled. The system performs well for all values of N and M. The best speedup is reached in the "64/64" benchmark where the final speedup is thirteen.

The limited speedup in the "256/32" benchmark could be a result of several workers that compete inside one computation instead of selecting their own sequences.

The most important improvement is not the implementation of g itself but the fact that the elements of the domain D can be treated as atomic data structures. When the elements of D were implemented by ordinary AKL terms the arithmetic operation g would first have to decompose three structures, check that their constituents actually were instantiated and, calculate and construct a new term to hold the result. The decomposition and construction of generic objects is of course far more efficient.

Benchmark	256/32	64/64	16/128	4/256	1/512
time ms.	590	650	680	820	1330

Table 9.18: Smith-Waterman, C

The execution time for a well written C program compiled with `gcc -O4` is shown in table 9.18. The increasing numbers for the C program can only be explained with a decreased cache performance. As the sequences grow larger the data structures do not fit into the cache. The second level cache miss rate grows from 0.02% to 0.4%. This is a penalty that is taken by the Penny system in almost all benchmarks.

The Penny program executes at about the same speed as the C program when three to six workers are used. The final execution time for the "64/64" benchmark is 259 milliseconds which is 2.5 times better than the C program. The "1/512" benchmarks is almost four times faster than the C program. The cache performance did not severely limit the performance even though we optimized the Penny program to mimic a native code compiler.

9.6.3 The Compiler

The Penny compiler can of course be executed in the Penny system itself. It consists of about 260 definitions using 1400 clauses, all in all about 3000 lines of AKL code including parser and output routines. It compiles itself in about thirty seconds using one worker. The parallel performance is initially good but then levels out at about thirteen seconds when six workers are used.

Figure 9.15 show the execution time (median of twenty runs) for the compiler compiling itself both with and without the output procedure included. As seen in the figure, the performance is initially good when the output routine is omitted. The speedup is 4.9 when six workers are used but then becomes drastically worse.

Let us find the reason for the behavior when the output routine is omitted. The statistics in Table 9.19 were obtained by logging all scheduling events. The number of stolen tasks, the average execution time per task and the average idle time between tasks was then calculated.

The number of scheduling operations increases dramatically as the number of workers are increased. When going form four to eight workers the number of stolen tasks is increased by almost two order of magnitudes. The average execution time for each stolen task is reduced by a factor of forty. These figures can be compared with the number of stolen tasks in the life benchmarks where the number of stolen task is only doubled.

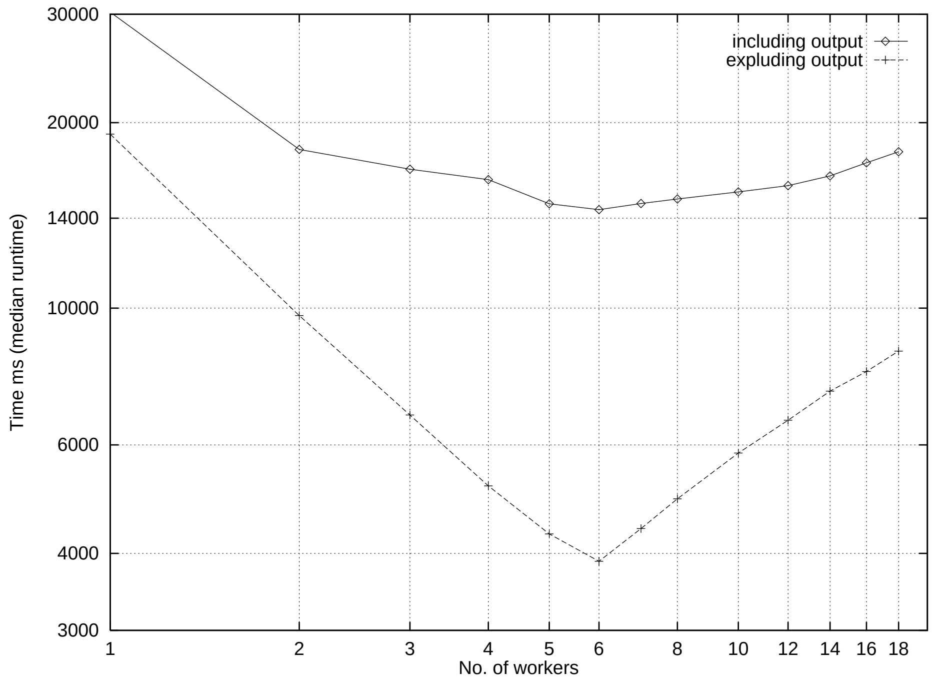

Figure 9.15: The compiler compiling itself

Workers	2	4	6	8	16
number of stolen tasks	389	986	9313	74427	233052
avg. execution time	50.90	20.89	2.50	0.53	0.38
avg. idle time	0.12	0.18	0.16	0.22	0.45

Table 9.19: Compiler excluding output

The idle time comparable to the calculated scheduling overhead in the matrix benchmark. This indicates that the workers find work relatively quick but the work they find is small grain.

The situation is improved by introducing a delay loop in the scheduler. If no work is found in a first sweep of the busy workers, a worker waits for a small amount of time before the next sweep is performed. As seen in Figure 9.16 the execution time is decreased when a longer delay is introduced. The delays used are approximately 0.04, 0.4, 4 and 40 milliseconds and is implemented as a dummy loop.

The same statistics were gathered for the benchmark using the longest delay. The figures are shown in Table 9.20. The total number of stolen tasks is reduced by more than a factor four. The average idle time is increased but this is compensated by the decrease in the number of scheduling operations. Actually few workers get caught in the delay loop. In the eight worker run only about two-hundred scheduling operations got caught. In the sixteen worker run about thousand operations involved the delay loop.

Workers	2	4	6	8	16
number of stolen tasks	273	657	5257	13798	49825
avg. execution time	68.35	30.40	4.07	1.77	0.77
avg. idle time	0.52	0.63	0.50	0.83	1.06

Table 9.20: Compiler, with delay, excluding output

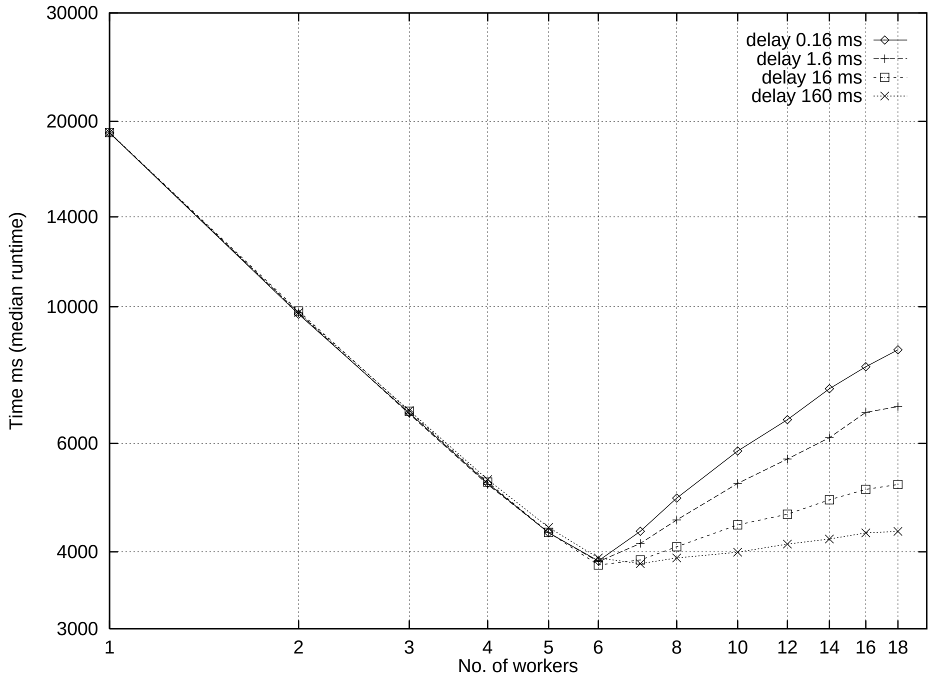

Figure 9.16: Compiler (excluding output) with various delays

The longest delay could of course be a problem for programs that depend on fast scheduling. In the matrix benchmark we were arguing that 0.08 milliseconds was a long time and here we want to add a delay that is several magnitudes higher. The delay loop is, however, only entered if no task has been found. The matrix benchmark actually runs well with all but the longest delay.

Including the output

The delay loop also improves the performance when the output routine is included. Table 9.21 shows the median execution time using the longest delay. The total execution time is reduced to 12.7 seconds and is then more or less constant regardless of how many additional workers that are used.

If we look at the execution time for one worker we see that the output routine is responsible for the last 11 seconds of the execution. Since two worker can do the actual compilation in 9.8 seconds, a third worker would be able to do the output

Workers	1	2	4	6	10	16
including output (s):	30.2	17.6	14.8	12.7	12.9	12.6
excluding output (s):	19.2	9.78	5.24	3.91	4.00	4.29

Table 9.21: Compiler with delay

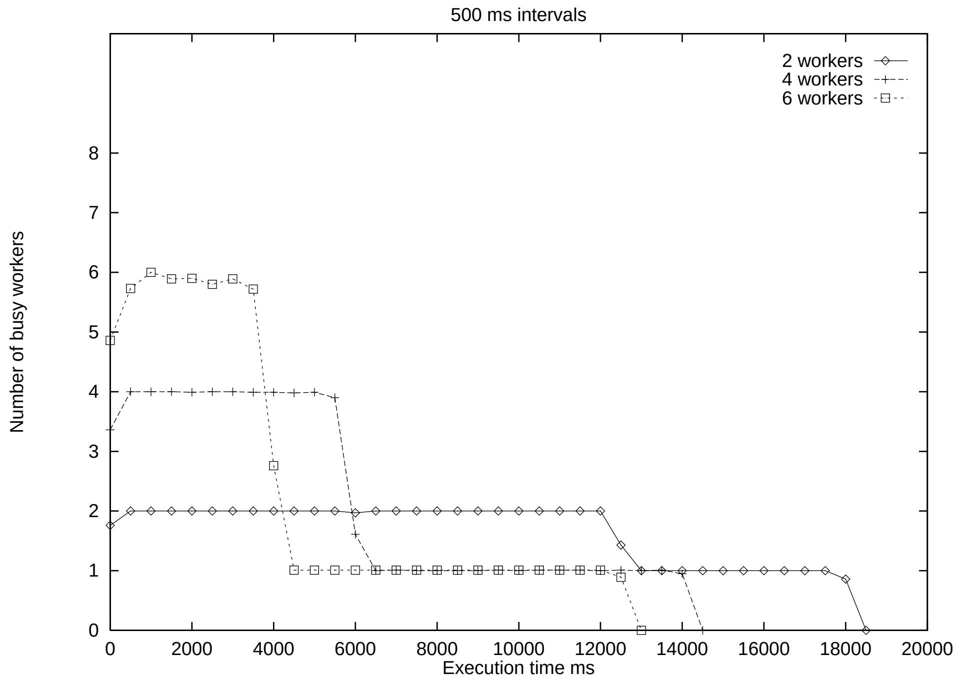

Figure 9.17: Number of busy workers during an execution

in parallel thus completing the whole execution in about eleven seconds. The minimum execution time is however 12.6 seconds. This is more than a second from the theoretical minimum.

If we look on the average execution time for each stolen task the figures does not indicate that the number of scheduling operations would cause a problem. The scheduling operations are few and the average execution time for each stolen task is high.

The amount of parallelism during the execution is however not constant. In Figure 9.17 the average number of workers that are actually busy (500 ms intervals) during the execution is shown for three runs with 2, 4 and 6 workers. As clearly seen the parallelism is high in the initial phase of the execution but when the actual compilation is completed only one worker at a time is active.

We can examine the execution in more detail if we plot all scheduling operations performed. In Figure 9.18 the four diagrams show the steal operations performed by each worker during the execution. The diagram is a state diagram where each

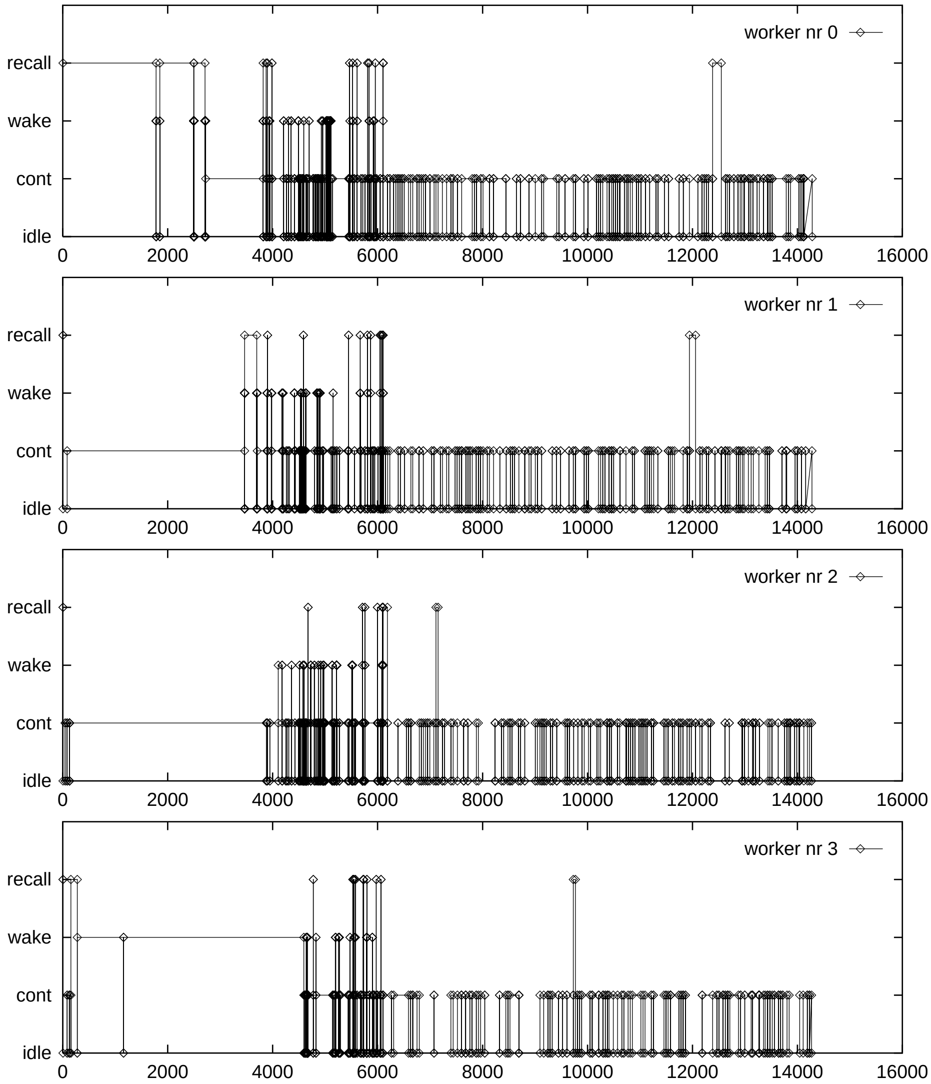

Figure 9.18: Stealing operations performed

point is a transition to or form the idle state. A worker that steals a task stay in the state of the stolen task until it returns to the scheduler.

The figure shows that all workers find profitable work during the first four seconds. They then take care of suspended computations for about two seconds. The actual compilation phase is then completed. During the last nine seconds they all fight over the same task, the continuation task that outputs the compiled code to a file.

Although the average number of scheduling operations is low the congestion during the last phase of the execution is a problem.

The lack of control

There is however a more serious problem in this program. Remember that the output routine takes about eleven seconds and since the compilation is done in 9.8

seconds by two workers the whole execution should be done eleven seconds by three workers. The execution time for three workers is 15.3 seconds. Why?

The output routine must, to minimize the execution time be scheduled as soon as data is available. The scheduler has of course no knowledge of this and schedules work that is not critical i.e. the compilation. All workers could be assigned to do the compilation while no one is scheduled to the output routine.

We are actually lucky that the output routine is scheduled at all, since the compilation phase can keep six workers busy. In a six worker run, all six workers could be assigned to the compilation and only when this work was exhausted would the output routine be scheduled. This would mean that the six worker run would complete the whole execution in $3.9 + 11.0 = 14.9$ seconds instead of the actual 12.7 seconds.

The good thing is that the output routine is in practice scheduled. The bad things is that if it was not, there would be little we could do about it. By making parallelism transparent to the programmer we have also removed all possibilities to control it. This is a big drawback if one sees the system as a parallel programming language.

9.7 Conclusions

Implicit parallelism works well if enough large tasks can be found. In a program with limited parallelism, compared to the number of workers, the workers tend to divide the available work into subtasks that are too small. Note, that the observed deficiency is a result of too many workers. As shown in Figure 9.16, a program can show very good speedup up to a certain point. Only when more workers are added will the reduced task size become a problem.

The cache performance of a parallel system is crucial. We have shown that the cache performance can be the limiting factor in programs with intensive inter-process communication. A cache miss rate of 1% is sufficient to limit the speedup. The cache miss-rate is a problem in the garbage collector since it has to traverse the whole execution state. A miss-rate of 11% was observed during garbage collection.

If parallelism is to be exploited in non-deterministic programs the system must be able to handle speculative work. Even if the system can limit the amount of speculative work the increased execution state will cause a problem. If the execution state grows with the number of workers then the garbage collection time will be constant even for the parallel garbage collector.

Chapter 10

Related Work

IN THIS CHAPTER I compare the Penny system to other parallel implementations of logic programming systems. Rather than doing an extensive performance comparison, I highlight the differences.

10.1 Basic execution speed

Comparing the performance of different systems is problematic since they run on different platforms and/or implement different languages. Nevertheless it is important to know the rough execution speed of the system. I have therefore included a comparison with SICStus Prolog v3 and the AGENTS 1.0 systems. SICStus Prolog is a well known sequential system that is easily available and the AGENTS system is the sequential implementation of AKL from which Penny was developed.

The benchmarks that I use do not test the suspension mechanism, garbage collection, deep computations, or non-deterministic execution. The benchmarks test only raw execution speed on simple recursive programs.

The benchmarks are:

mastermind: The mastermind puzzle by E. Tick, using two guesses and three colors.

matrix: A 500×500 matrix (in this test filled with ones to avoid using big-nums or floats) multiplied by a vector.

hanoi: The towers of Hanoi puzzle by E. Tick using seventeen bricks.

kkqueen: The candidates/non-candidates queens program by K. Kumon/E. Tick, using 9 queens.

tsp: A near-optimal solution to the traveling salesman problem by M. Veanes, using twenty-four nodes.

System	Penny	AGENTS 1.0	SICStus v3(em)
hanoi	2.9s (1.0)	2.7s (0.93)	1.6s (0.55)
kkqueen	2.7s (1.0)	3.1s (0.87)	1.0s (0.37)
tsp	1.7s (1.0)	1.7s (1.0)	0.63s (0.37)
mastermind	2.6s (1.0)	2.2s (0.85)	0.65s (0.25)
matrix	2.5s (1.0)	2.5s (1.0)	1.3s (0.52)

Table 10.1: Sequential execution speed

System	Penny	SICStus v3(n)
hanoi	2.9s (1.0)	0.43 (0.15)
kkqueen	2.7s (1.0)	0.36 (0.13)
tsp	1.7s (1.0)	0.23 (0.14)
mastermind	2.6s (1.0)	0.27 (0.10)
matrix	2.5s (1.0)	0.96 (0.38)

Table 10.2: Penny vs. SICStus v3 native

Table 10.1 shows the relationship between Penny, AGENTS 1.0 and emulated SICStus v3 [1] on a SunSPARC-20. The Penny system is the slowest of the three but the AGENTS system is less than 15% faster. The SICStus v3 system is more than twice as fast as the Penny system. The difference is smaller in the hanoi and matrix benchmarks, larger for the tsp and kkqueen benchmark and largest for the mastermind benchmark.

The large difference between Penny and SICStus is mainly due to the poor decision code and poor register allocation generated by the Penny compiler. If a guard contains an equality test or an arithmetic operation a deep guard is created. This can be avoided if the correct switch instructions are generated. In an experiment where the abstract machine code of the kkqueen program was hand optimized, using the available instruction set, the execution time was reduced to 2.0 seconds, i.e. by 26%.

Preliminary results from the new AGENTS compiler developed by Per Brand show figures that are as good as the figures for SICStus v3 [10]. This indicates that a better compiler and runtime system can speedup the sequential performance of the Penny system by a factor of two.

Table 10.2 shows the execution time using the native code compiler of SICStus v3. The native code compiler of SICStus v3 is for these benchmarks more than twice as

[1] Using compact-code, wall-time, patch 5, November 1996

Processors	1	2	4	6	8	12	16
MUSE	2.5s	1.5s	1.0s	0.79s	0.66s	0.56s	0.49s
Penny	22s	11s	6.0s	4.1s	3.1s	2.5s	2.5s
Ratio Penny/MUSE	8.8	7.3	6.0	5.2	4.7	4.5	5.1

Table 10.3: Ten queens, all solutions, time in seconds

fast as the emulated system and thus more than five times faster than the Penny system.

Notice that these benchmarks test only forward execution without any overhead for garbage collection etc. Comparing systems on the basis of these figures is like looking at the speedometer of a car; pointless but still a popular hobby.

10.2 Parallel implementations of Prolog

Parallel implementations of prolog have been divided into or-parallel and and-parallel systems. Only in the last years have systems been built that include both forms of parallelism while maintaining the Prolog execution model [24].

10.2.1 Or-parallelism

There are two challenges when building an or-parallel Prolog system, maintaining the multiple binding environments and minimizing speculative work [41, 4]. The binding schemes used in these systems are quite different from the scheme used in the Penny system. This is only natural since the execution state of a deep AKL program is very different from a Prolog execution.

In the Penny system the use of an or-parallel binding scheme is avoided by creating an explicit copy in each choice splitting operation. This is of course expensive and induces a cost that is hard to justify. To estimate the overhead I have compared the Penny system to MUSE, an or-parallel Prolog system available with the SICStus v3 system [63].

Table 10.3 shows the performance using a ten-queens benchmark on an eight processor SPARCCenter-2000. In this benchmark the MUSE system outperforms the Penny system by almost a factor of nine. This is not surprising since the MUSE system uses backtracking whereas the Penny system uses copying to perform non-deterministic computations. Notice however that the difference decreases, an indication that the Penny system scales up well when the number of processors are increased.

Since the benchmark searches for all solutions to the puzzle no speculative work is performed. In a benchmark where the leftmost solution is requested the MUSE systems will outperform the Penny system even more. Notice however that no intelligent constraint handling is performed nor is concurrency used in this version of the queens program. The Penny system will not make up for the copying overhead with a smaller search space or increased parallelism. By using a benchmark where the Penny system could benefit from its more advanced search strategies, Penny would be put in a better light but this would be to compare apples and pears.

The MUSE system is, like the Penny system, an emulated system. The native-code SICStus compiler will run the same benchmark in 1.5 seconds on the same machine using one processor.

10.2.2 And-parallelism

The execution schemes for and-parallel Prolog differ mainly in how dependent computations are handled. The first schemes implemented various forms of independent and-parallelism. The problem is how to detect that computations are independent. Compile-time analysis is sometimes not enough and must therefore be complemented with runtime tests [29].

When dependent computations are executed in parallel the big problem is how to handle non-determinism. Various solutions have been proposed that vary in their degree of providing unrestricted parallelism [53]. The Dynamic Dependent And-parallel Scheme (DDAS) proposed by Kish Shen is so far the most promising[61]. In this scheme the system keeps track of variables that are shared between workers. The worker that is working on the leftmost goal has priority to bind a shared variable, other workers will suspend when they try to access the variable. If the binder of the variable later has to backtrack it will signal the consumers of the variable to redo their computations.

An implementation of the scheme (DASWAM) shows good parallel performance. [62]. This system so far relies on user annotations to collect information about shared variable, but this can in the future be provided by a compiler.

A radical approach was taken by the Reform group at Uppsala University [9]. Their approach was to limit the system to handle only and-parallelism generated in linear-recursive deterministic procedures. The system can still handle non-deterministic computations but this will limit the parallelism. The restriction is well chosen; deterministic procedures are the main source of parallelism in Prolog programs. By avoiding the problem of handling non-determinism the system can compete with optimized sequential systems. The Reform system is the fastest and-parallel system for the type of programs that complies to the restrictions [39].

10.2.3 A combination of the two

Combining and- and or-parallelism is tricky. Andorra-I is one system that uses the Basic Andorra Model to solve the problem [54]. Deterministic goals are allowed to proceed while non-deterministic goals are temporary suspended. When a non-deterministic step is performed the resulting branches can be executed in parallel. The execution model improves the ordinary Prolog search strategy, but it is still inferior to finite-domain constraint based systems [57].

A problem with the Andorra-I approach is that it does not provide the programmer with the tools to guide the search. It is up to the compiler to detect determinate goals. There is no clear notion of a guard as in the AKL. Nor is it possible to encapsulate non-deterministic computations, a deficiency that makes it hard to design reactive systems that also exploits the non-determinism.

In ACE, a system based on a parallel copying execution scheme, each node in an or-parallel execution can be executed using and-parallelism [24]. The scheme combines the execution mechanism of MUSE and &-Prolog. The combination of the to models introduces implementation complexity. The exploited parallelism is similar to the Andorra-I system, but the ACE system does not change the execution model of the language. The system still has to prove that it can compete with a state-of-the-art sequential system, and that the extra complexity is worth the gain in parallel performance.

10.3 Concurrent logic programming

The research in concurrent logic programming languages started as a means for parallel execution. Although the concurrency itself is as important as the the gain in execution time, the interest was directed to parallel implementations.

In the quest for performance the developers gradually removed properties of traditional logic programming languages. Non-determinism was abandoned at an early stage and the systems were then restricted to handle only flat guards [58, 68]. Some systems went even further and removed general unification, disallowed multiple readers of variables, and included imperative variables [22, 21].

10.3.1 KLIC

One of the systems that kept the logical variables in the language is KL1. The languages was developed at ICOT in the Fifth Generation project and was used both to implement the operating system and various high-level application oriented programming languages [71]. The KLIC implementation of KL1 is efficient and supports shared-memory architectures as well as message passing and distributed architectures [14].

System	Penny	KLIC
hanoi	2.9s (1.0)	0.59 (0.20)
kkqueen	2.7s (1.0)	0.22 (0.08)
tsp	1.7s (1.0)	0.64 (0.39)
mastermind	2.6s (1.0)	0.13 (0.05)
matrix	2.5s (1.0)	0.26 (0.10)

Table 10.4: Penny vs. KLIC

The KLIC compiler compiles KL1 programs into C. The basic execution speed of the KLIC system is comparable to native-code system. The system, is as shown in Table 10.4, much faster than the Penny system.

When running on one processor the KLIC system has an advantage over the Penny system. In the KLIC systems, variables shared between workers have a special tag. As long as un-shared variables are handled no locking is needed. The Penny system does not know whether a variable is shared or not and has to do the locking even if only one worker is doing the execution.

The main deficiency of the Penny system compared to the KLIC system is Penny's naive way of handling built-in procedures such as arithmetic. The Penny system must also handle many simple guards as deep computations since the compiler will not detect the possibility to make a flat suspension.

10.3.2 Explicit parallelism

The KL1 uses explicit annotations in a program to direct where and how parallel execution should be performed. The semantics of the program however does not change when parallel annotations are added. Explicit annotations are necessary in a program when the target is a message passing machine or a cluster of workstations. In a shared-memory implementation the programmer can aid the runtime system by dividing the work into larger tasks. Explicit annotations can mean better performance but does make the programming task more difficult.

To see if explicit annotations can help in the distribution of work the life benchmark was executed using KLIC and Penny. This benchmark could benefit from better locality in the computation to reduce the cache miss rate.

The KL1 version of the game of life is a reduced game where each cell has only four neighbors. The grid is divided into clusters and the clusters are then distributed on the available processors. I ran the experiment with a 40×40 grid divided into sixteen clusters of 10×10 cells each. The division allows an even distribution of clusters on the available processors.

Workers	1	2	4	8	16
KLIC	10.3s	5.4s	2.9s	2.5s	2.2s
Penny	17s	9.6s	5.2s	3.1s	2.1s
Ratio Penny/KLIC	1.65	1.78	1.79	1.24	1.00

Table 10.5: The game of life (time in milliseconds)

In the Penny version no annotations are necessary to parallelize the program. The program is simplified since there is no need to divide the program up into clusters, all cells are treated equally. A benchmark like the life program could be the death of an implicit parallel system like Penny, but it turns out that the Penny system performs well.

Table 10.5 shows the performance of the KL1 life benchmark. The timings were the best in ten runs. For the KLIC system the ten runs were all done in one execution to avoid system time overhead in the initialization of heap areas. In the Penny system heap allocation is done at start up and is not included in the reported execution time.

The figures show that the Penny system, for this benchmark, is only twice slower than the KLIC system. The parallel performance is however better for the Penny system. When sixteen processors are used the Penny system is even with the KLIC system.

There are several possible reasons for the life benchmark showing good results using the Penny system.

- The KLIC system is much faster on instruction decoding since it does not have the overhead of an emulator. The instruction decoding is not important for this benchmark since a large part of the execution time is spent in other parts of the system.
- Binding shared variables (shared between workers) is in the KLIC system considerably more expensive than binding non-shared variables. In the life benchmark about 10% of the communication (through variables) is performed with shared variables.
- The explicit distribution could be harmful since a worker will have to wait for other workers to perform their tasks before the next generation of border cells can be computed.

10.3.3 Deep guards

Deep guards were used in the early committed choice languages for example Guarded Horn Clauses (GHC), Concurrent Prolog (CP) and PARLOG [15, 58, 70, 66]. These languages used deep guards for complex deterministic tests. In CP the guard computation was allowed to make local bindings to external variables. This introduced a complexity in the implementation.

In PARLOG the local bindings of external variables was restricted to the *output arguments* of the definition and these where only made visible at the time of commitment. In GHC the problem with local bindings was solved by simply disallowing any bindings of external variables. This was the most elegant solution in terms of the number of language constructs and its implementation.

Once the guards had been reduced to deterministic test that could not hold any local variables the need for a deep guard is of course limited. It is even possible to transform almost any GHC program to a flat program that does not use any deep guards. The deep guards were also later removed from the languages resulting in the flat committed choice programming languages.

10.3.4 Including non-determinism

Non-determinism is included in PARLOG but the language makes a distinction between *single-solution* relations and *all-solutions* relations [15]. The single-solution relations provided concurrency whereas the all-solutions relations provided the non-determinism. A PARLOG program is at the top-level executing in single-solution mode but can through special constructs call goals defined by all-solutions relations.

The two worlds cannot share variables so the all-solutions goal must be completely instantiated apart from existentially quantified variables. This is required in order to give the system a clear operational semantics. Since the non-determinate computation does not share any variables with the rest of the system it can be implemented using ordinary Prolog techniques.

The AKL does not place any such restrictions on the program or use of non-determinism. The problem with shared variables is solved by encapsulating non-determinism in guards and using the notion of stability.

10.4 Parallel implementations of AKL

This section describes the tw other parallel implementations of AKL, namely ParAKL and DAM, and how they differ from the Penny system.

10.4.1 ParAKL

ParAKL is the parallel implementation of AKL developed by Remco Moolenaar at the University of Leuven [47]. The implementation was developed using an early

System	Penny	ParAKL
hanoi (14)	0.72s (1.0)	2.4s (3.33)
matrix (250)	0.61s (1.0)	1.5s (2.46)

Table 10.6: Sequential execution speed, ParAKL

version of the AGENTS system. The version that the Penny system is based on is about three times faster than the ParAKL system.

Table 10.6 shows figures for the sequential execution time on a SunSPARC-20. The benchmarks are reduced versions of the benchmarks used in the previous sections. The Penny system is for these small benchmarks as expected a factor three faster than the ParAKL system. No comparison has been done for parallel execution since the ParAKL system does not run in parallel on the SPARCCenter-2000.

The implementation differs from the Penny system mainly in the handling of binding environment. The binding scheme in ParAKL is built using the PEPSys hashing scheme [7]. The workers maintain a shared tree of hash tables that are updated to reflect the binding environments of the different and-boxes.

The hashing scheme is more complicated than the explicit binding scheme used in the Penny system. The scheme was originally developed for an or-parallel Prolog implementation, a system where local bindings of variables must be highly optimized. An AKL execution however is different from an or-parallel execution, as shown earlier in Section 7.6.

The ParAKL implementation exploits parallelism in two ways. It executes goals in an and-box in parallel and it has a parallel implementation of the choice splitting rule. This is different from the Penny system where both and- and or-parallelism can be exploited but the copying procedure itself is a sequential operation.

The speedup reported is not convincing. Even simple recursive benchmarks, e.g. matrix multiplication, show low parallel performance (speedup 8.99 using 20 CPU). The performance of the non-deterministic benchmarks are of course limited since only the choice splitting rule itself is parallelized. The system is developed and evaluated on a SequentSymmetry, an older architecture that does not represent modern cache based machines.

In his thesis Moolenaar describes some implementation techniques to improve non-deterministic execution by more intelligent selection of candidate clauses. Using these techniques one can get good results for puzzle programs but the system can not compete with a finite domain constraint systems.

10.4.2 DAM

The DAM is a parallel implementation of AKL by Doug Palmer at the University of Melbourne. The binding scheme used in DAM has many similarities with the scheme used in the Penny system.

One difference is the way the locality of variables is determined. In the DAM all variables that are local to an and-boxed are linked in a list. This is done in order to find the variables and redirect their environment pointer when the and-box is promoted. The implementation introduces a sequential component in the promotion operation, a component that is proportional to the number of created variables in the node. The promotion itself need not be expensive since the list should contain few unbound variables when it is time for promotion.

The system has been developed on a SPARCServer but so far, no reports on the performance of the system have been published.

10.5 Other related systems

Implicit parallelism is also a hot topic in functional programming research. The functional systems are easier to implement since they do not handle logical variables and consequently not encapsulated computation spaces. The flat sub-set of AKL does however have many similarities with the functional programming languages.

The Multilisp system was one of the first systems that exploited implicit parallelism [26]. It is a system that is similar in its approach to the Penny system in that it created the tasks on demand.

The pH system is a parallel version of Haskell developed by Arvind's group at the MIT Lab for Computer Science, and Nikhil at the DEC Cambridge Research Lab [1]. The pH system is developed using experience gained in the work on Id, a functional language designed for data-flow machines [49].

One interesting feature in pH is the so-called *I-structure*. The data type is a record that can hold holes. The holes can be assigned a value but only once. The I-structures are thus closely related to the variables used in logic programming languages. Another interesting feature in pH is the *M-structure* [8]. The structure allows multiple atomic updates and is similar to the *cell* construct provided in Oz in combination with a logical variable [64]. The M-structure solves the same problem as the port construct in Penny.

Chapter 11

Summary

As explained in the introduction, this dissertation presents the *design*, *implementation*, and *evaluation* of a system that exploits fine-grain implicit parallelism in a concurrent constraint programming language.

The implementation is the first system that exploits and-parallel execution of deterministic procedures and or-parallel execution of encapsulated non-deterministic search.

11.1 The design

It is always important to have a clear understanding of the semantics of a language in order to implement it. This is especially true when designing a parallel implementation. Properties of language constructs that have not been precisely described in a computational model can cause inconsistencies in the implementation. Inconsistencies might lead to undefined but predictable behavior in a sequential implementation but a parallel implementation will most certainly be unpredictable.

The AKL language is described by a computation model that is independent of the particular order of execution. The computation model is the level at which language constructs should be defined. Language constructs that can be described at this level are safe to introduce while constructs that cannot be defined, almost certainly will cause unpredictable behavior in a system.

The execution model defines the operational behavior of the Penny system. The system implements a subset of the computational model. The execution model describes the order in which rewrite operations should be performed. The model is designed in a way that allows exploitation of implicit parallelism while keeping the sequential performance.

The execution model is lazy, i.e. suspended goals are not woken as soon as the necessary information is available. This means that a worker that is producing data can continue to do so even though suspended goals are ready to consume the data. A lazy scheme is often regarded as inefficient since the execution state can become unnecessarily large.

I have shown that an eager scheme leads to unacceptable behavior in a parallel system that is based on implicit concurrency. The execution state will for certain programs grow as suspended goals are woken. A lazy execution model is for these programs more predictable. The lazy approach also improves the locality of execution. Workers continue with one part of the execution instead of moving around. This is important in order to keep the sequential performance of the system.

The execution model is not fair, i.e. the execution does not guarantee that possible reductions eventually are done. This means that goals might be indefinitely suspended even though a rewrite operation can be performed. This was an intentional design principle. Combining implicit concurrency, parallelism and fairness, on the same level, is hard to manage. In a deep concurrent constraint system it would also be inefficient.

Fairness must be considered in order for the language to be practically useful as a tool for building reactive systems. The fairness property should however not be guaranteed on the same level as language concurrency. A concept at a higher level, call it a process, should guarantee fairness but this concept should also handle allocation of resources, priority, mobility etc. Within a process one could have implicit lazy concurrency. This would allow parallelism to be extracted within each process.

11.2 The Implementation

The implementation is stable and very robust. It is possible to run larger benchmarks including real-life programs. In the parallel logic programming community it is one of few experimental systems that manages to run its own compiler. Running larger programs is necessary in order to reveal the limitations of the system.

11.2.1 The emulator

The runtime system is a WAM style emulator. It is implemented as a thread-code emulator with a basic performance of about half that of state-of-the-art emulators.

The weakest part in the system is the compiler. It does not handle flat suspensions for anything but the simplest examples, nor does it optimize the register allocation. The improved compiler that is under development has shown an improvement by a factor of two to three. I have shown that the overhead in the emulator induced by the parallel system is small. There is nothing that prevents the current system to make use of the improved compiling technique. This will, it is hoped, be proved in the future.

11.2.2 The binding scheme

The binding scheme is the most efficient binding scheme presented for parallel implementations of deep concurrent constraint systems. The binding scheme takes care of local bindings in a hierarchy of constraint stores. The alternative bindings that result from non-determinate execution is handled by explicit copying.

Local bindings to external variables are explicitly recorded in a list that is associated to the constraint store. Finding the local value of an external variable can result in the traversal of several lists. The number of traversals and the length of the lists are in practice small. They are not dependent on the size of the program nor the size of the problem. The small number of levels and the few local bindings on each level has to do with the programming style. The binding scheme is justified by empirical data from a set of benchmark programs.

The binding scheme also keeps track of suspended computations and the stability of local computations. The stability detection is important in a parallel system since it allows parallel execution of non-deterministic computations. A system that detects stability only partially is limited, not in functionality but in parallel performance.

An additional binding scheme could be used in the implementation of the non-determinate choice spitting operation. The system would then have two orthogonal binding schemes. One scheme that keeps track of the levels in the AKL execution and another scheme that keeps track of alternative bindings after choice splitting. Local variables would then be divided into young and old variables i.e. variables that have been created after or before the last choice splitting operation. This would result in a complicated scheme that would be difficult to parallelize.

11.2.3 The scheduler

The design of the scheduler was done at first with simplicity in mind. Since it was important to keep the performance of the sequential system the scheduler was designed in such a way that it would only be used if needed. Very little overhead is put on an executing process while an idle process has to do all work by itself.

Apart from registering as an idle worker, there are no central resources. An idle worker scans the busy workers for available tasks. If a task is found the stacks of the busy worker are locked. The busy worker can continue to work but cannot remove items from the stacks. The idle worker steals tasks from the bottom of the stacks in order to minimize the interaction with the busy worker.

Any task can be stolen. The scheduler is able to steal a task that belongs to deep computations. This is vital for non-determinate computations, where all tasks are found in deep computations. It is less important for deterministic computations. The scheduler could be simplified, and possibly be more efficient, if scheduling was limited to the upper level.

11.3 Evaluation

The system is robust and gives predictable performance even on a loaded machine. The parallel performance of the system is very good. For simple deterministic benchmarks a speedup of fifteen can be achieved using twenty processors.

11.3.1 Granularity

The scheduling overhead can be a problem for programs with fine-grain tasks. If each task requires a global scheduling operation the speedup is limited. In a benchmark with a task size of 0.26 milliseconds, the speedup using two workers was limited to 1.5. When the task size was increased to 2.1 milliseconds the initial speedup was 1.9. Since the Penny system executes at about 260 Klips, the 0.26 milliseconds is equivalent to about 70 logical inferences. This is roughly the limit of the task size.

Another threat to performance is the spawning of tasks. If there is only one task in the system, and this task holds the key to the parallel execution, there is a sequential thread in the system. The turn-around, i.e. the time from the moment one worker steels the task until another worker can steal it, sets a limit on the minimal execution time. The turn-around time also sets a limit on the number of workers that can productively take part in the execution. In the benchmark with 0.26 millisecond tasks, the limit was six workers.

11.3.2 Cache performance

Cache performance has been investigated by using the instruction level simulator SimICS. This was the first study that described the cache performance, at the level of machine instructions, of a logic programming system.

Cache performance cannot be underestimated. The Penny system was design with cache performance in mind but globally shared data structures impose a high cache miss rate. The miss rate increases with the number of processors and is about 1% when sixteen workers are used.

The cache performance is the limiting factor for benchmarks with high ratio of communication. This is very interesting since it changes the focus for optimizations. So far the attention has been directed to instruction decoding, locking schemes, or scheduling techniques. The results show that the cache performance is as important to optimize.

11.3.3 Garbage collection

Garbage collection is a very important but often excluded part of a parallel implementation. A parallel system is never better that its garbage collector. If the garbage collector is sequential the overall speedup is limited. Parallel garbage collection is a minimal requirement but does not solve the whole problem. A stop-and-copy garbage collector is for example limited by the largest structure.

The effect can be reduced through a generational garbage collector, where large data structures hopefully can be moved to an older generation. This reduces the overall garbage collection time but it is only a temporary solution. A definite solution to the problem will probably include a scheme that allows runtime garbage collection.

11.3.4 Non-determinism

Exploiting parallelism in a non-deterministic computation is difficult. It is as shown easy to compute all the solutions for a problem but if only one solution is needed the performance depends on how good the system is on limiting the amount of speculative work. The Penny system is restrictive when it evaluates guards in parallel; but, evaluates the alternatives in parallel after choice splitting operation.

To provide more predictable behavior the system could be limited. One alternative is to only execute aggregate constructs (bagof etc) in parallel. The parallelism would then be exploited only when all solutions are requested. This can improve the performance on programs where more than one non-deterministic process is active.

11.4 Discussion

Parallelism is an important component in a high performance programming system, but it is important to question its usefulness.

11.4.1 The user

Among the users of declarative programming languages today, I don't think anyone has had reduced execution time as the number one reason for selecting the language. There are other features of the language and system that are more important. Ease of programming, debugging, and inter-operability have probably been more important factors. The execution time is important, but it will not compensate for a bad programming environment or lack of functionality.

On the other side of the spectra we have the users of explicit parallel systems. This group of users has probably had reduced execution time as their number one priority. If reduced execution time is important then the lack of functionality can be accepted.

A parallel system will be accepted either if provides the same ease of programming as the declarative systems or if it can outperform any parallel system.

11.4.2 The machine

The second question is what the target machine will look like. We can divide the machine into groups depending on the number of processors: low-end machines with up to four processors, midrange machines with up to 16 or 32 processors and the high-end servers with even more processors.

If the target of the system is the low-end group, the parallel system will have competition from highly optimized native-code generating compilers. There is no contradiction in building a highly optimized parallel system. But this might require more resources than can be justified by the limited gain in performance. Note that an unoptimized parallel system has small chances to be competitive for this range of processors.

For the midrange machines, parallelism is more attractive. Even the Penny system, in its current unoptimized state, is very competitive compared to optimized sequential systems. If this is the target there is no question of the benefits of developing a parallel system.

In the high-end segment there is no question that a parallel system can outperform a sequential system. The question here is rather what applications that can benefit from these machines.

11.4.3 The law

Amdahl's law states that no matter how much you improve one part of a program, the other part will limit the overall speedup. In parallel programming it is the sequential thread in a program that sets the limit. If there is a sequential thread in the program, that accounts for 20% of the execution time, the speedup is limited to a factor of five.

The programs that can benefit from a machine with more than 32 processors must not have a sequential thread that is larger than 3% of the total execution time. The more processors that are used, the smaller the sequential thread must be.

The consequences of Amdahl's law is often used as an argument against the possibilities of parallel programming. There are few applications that have are free of sequential dependencies. This is partly correct, but the argument is based on the assumption that the programs will look the same but there is reason to believe that the programming style will change. Instead of writing one complicated procedure it is will be easier, and equally efficient, to write several simple procedures that solves the same problem. When a programmer is faced with a system, where adding parallel tasks does not cost anything, he or she will make use of the resources. The execution power of multiprocessors will be filled like the Gigabyte hard-disk on you PC.

11.4.4 The future

I believe that the parallel systems of tomorrow will use implicit parallelism. The implicit parallelism extracted from a program will simply be sufficient to achieve close to maximal performance. The reason why implicit parallelism is regarded as a hard problem is more related to the deficiencies in sequential programming languages than to some inherent property in the approach.

The first generation will probably use explicit concurrency as the source of parallelism. When this is not sufficient the parallelism in each thread will be exploited. When parallelism is exploit within a thread a declarative programming language is very attractive.

The parallel performance can be the main driving force for declarative systems. The systems will be developed and used not because of their theoretical advantages but simply because they will provide better performance. My hope is that this dissertation is a step in that direction.

Epilogue

THE STARTING POINT is hard to define but the end is not. I have a dead-line tomorrow afternoon that must be kept. During my time at SICS I have so far never been able to hand in a paper a day in advance and this dissertation will not be an exception. Since I write tomorrow you know that I'm writing this in advance, that is today. But I ensure you that there will be last time changes tomorrow (I told you, today is tomorrow). Then, at two o'clock, I'm going to curse the laser-writer and beg DHL to have another cup of coffee... hmmm, since you are reading this I guess everything went OK, no worry... but what if I'm only writing this and no one will ever read it... worry!

⋆ ⋆ ★ ⋆ ⋆

References

[1] S. Aditya, Arvind, J. Maessen, L. Augustsson, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In *Proceedings from the Haskell Workshop (at FPCA 95)*, June 1995.

[2] H. Aït-Kaci. *The WAM: A (real) tutorial.* MIT Press, 1991.

[3] K. A. M. Ali. A parallel copying garbage collection scheme for shared-memory multiprocessors. *New Generation Computing*, 13(4), December 1995.

[4] K. A. M. Ali and R. Karlsson. The MUSE or-parallel Prolog model and its performance. In *North American Conference on Logic Programming.* MIT Press, October 1990.

[5] G. A. M. A. Atlam. *Parallel Garbage Collection in a Multiprocessors Implementation of a Concurrent Constraint Programming System.* Phd thesis, Menoufia University, Egypt, January 1997.

[6] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[7] U. Baron and et.al. The parallel ECRC Prolog system PEPSys: an overview and evaluation results. In *Proceedings of the International Conference of Fifth Generation Computer Systems*, pages 841–850, 1988.

[8] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings on Functional Programming and Computer Architecture*, pages 28–30, March 1991.

[9] J. Bevemyr, T. Lindgren, and H. Millrot. Reform Prolog: the language and its implementation. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, Budapest, Hungary, 1993. The MIT Press.

[10] P. Brand. Performance of the AKL compiler. personal communication.

[11] P. Brand. Decision graph compilation. In *Proceedings of the Twelfth International Conference on Logic Programming.* MIT Press, 1995.

[12] B. Carlsson. *Compiling and Executing Finite Domain Constraints.* Uppsala thesis in computing science 21, SICS dissertation series 18, Uppsala University, SICS, 1995.

[13] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[14] T. Chikayama. KLIC: a KL1 Implementation for Unix Systems. *New Generation Computing*, 12(1):123–124, 1993.

[15] K. Clark and S. Gregory. *PARLOG: Parallel Programming in Logic*, chapter 3, pages 84–139. MIT Press, 1987.

[16] J. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors*. Phd thesis, Heriot-Watt University, 1988.

[17] I. de Castro Duatra. Strategies for scheduling and- and or- work in parallel logic programming systems. In M. Bruynooghe, editor, *Proceedings of the 1994 International Logic Programming Symposium*, Ithaca, 1994. ALP, MIT Press.

[18] S. Debray and M. Jain. A simple program transformation for parallelism. In M. Bruynooghe, editor, *Proceedings of the 1994 International Logic Programming Symposium*, Ithaca, 1994. ALP, MIT Press.

[19] A. K. Dewdney. How to resurrect a cat from its grin. *Scientific American*, pages 124 – 127, September 1990.

[20] J. R. Ellis, K. Li, and A. W. Appel. Real-time concurrent collection on stock multiprocessors. In *SIGPLAN' 88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988. Also Digital SRC Research Report number 25.

[21] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):51–66, 1992.

[22] I. Foster and S. Taylor. Strand: A practical parallel programming language. In *Proceedings of the North American Conference on Logic Programming*, 1989.

[23] T. Franzén. Some formal aspects of AKL. SICS Research Report R94:10, Swedish Institute of Computer Science, 1994.

[24] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: and/or-parallel copying-based execution of logic programs. In P. van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 93–109, Santa Marherita Ligure, Italy, 1994. The MIT Press.

[25] G. Gupta and B. Jayaraman. On criteria for or-parallel execution models of logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 737–756, Austin, 1990. ALP, MIT Press.

[26] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. In *ACM Transactions on Programming Languages and Systems*, volume 7, pages 501–538, October 1985.

[27] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions Programming Languages and Systems*, 7(4):501–538, October 1985.

[28] M. Hermenegildo and M. Carro. Relating data-parallelism and (and-) parallelism in logic programs. In *Lecture Notes in Computer Science, 966*, pages 27–41. Springer-Verlag, August 1995.

[29] M. Hermenegildo and F. Rossi. Strict and non-strict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *Journal of Logic Programming*, 22(1), January 1995.

[30] A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Computing*, 4(9):1030–1040, September 1993.

[31] R. Jain. *The art of computer systems performance analysis*. Wiley Professional Computing, 1991.

[32] S. Janson. AKL: A multiparadigm programming language. Uppsala Thesis in Computing Science 19, SICS Dissertation Series 14, Uppsala University, SICS, 1994.

[33] S. Janson and S. Haridi. Programming paradigms of the Andorra Kernel Language. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991. The MIT Press.

[34] S. Janson and J. Montelius. The design of the AKL/PS 0.0 prototype implementation of the Andorra Kernel Language. ESPRIT deliverable, EP 2471 (PEPMA), Swedish Institute of Computer Science, 1992.

[35] S. Janson, J. Montelius, and S. Haridi. *Ports for Objects in Concurrent Logic Programs*, chapter 8, pages 211–231. MIT Press, 1993.

[36] R. Karlsson. Parallelism in Smith-Waterman. personal communication.

[37] J. S. Larson, B. C. Massey, and E. Tick. Super Monaco: Its portable and efficient parallel runtime system. In *Lecture Notes in Computer Science, 966*, pages 527–538. Springer-Verlag, August 1995.

[38] B. Lewis and D. J. Berg. *Threads Primer*. SunSoft Press, 1996.

[39] T. Lindgren, J. Bevemyr, and H. Millrot. Compiler optimizations in Reform Prolog: Experiments on the KSR-1 multiprocessor. In *Lecture Notes in Computer Science, 966*, pages 553 –564. Springer-Verlag, August 1995.

[40] J. Loeckx and K. Sieber. *The Foundation of Logic Programming*. Pitman Press, Ltd., Bath, Avon, 1984.

[41] E. Lusk, D. H. D. Warren, S. Haridi, et al. The Aurora or-parallel Prolog system. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[42] P. Magnusson. A Design for Efficient Simulation of a Multiprocessor. In *Proceedings of MASCOTS*, pages 69–78, January 1993.

[43] P. Magnusson and B. Werner. Some efficient techniques for simulating memory. Technical Report T94:16, Swedish Institute of Computer Science, August 1994.

[44] P. Magnusson and B. Werner. Efficient Memory Simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium*, 1995.

[45] M. J. Maher. Logic semantics for a class of committed choice programs. In *Proceedings of the Fourth International Conference on Logic Programming.* MIT Press, 1987.

[46] T. Miyazaki, A. Takeuchi, and T. Chikayama. A sequential implementation of concurrent Prolog based on the shallow binding scheme. In *Symposium on Logic Programming*, pages 110–118. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, July 1985.

[47] R. Moolenaar. *The parallel implementation of the Andorra Kernel Language.* Phd. thesis, Katholieke Univeristeit Leuven, 1995.

[48] R. Moolenaar and B. Demoen. A parallel implementation for AKL. In *Lecture Notes in Computer Science, 714*, pages 246–261. Springer-Verlag, August 1993.

[49] R. S. Nikhil and Arvind. *Id: A language with implicit parallelism.* Elsevier Science Publishers, February 1990.

[50] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In *Proceeding of the Third Conference on Functional Programming Languages and Computer Architecture*, pages 113–133, September 1987.

[51] A. Podelski and G. Smolka. Situated simplification. In U. Montanari, editor, *Proceedings of the 1st Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 976, pages 328–344, Cassis, France, September 1995. Springer-Verlag.

[52] A. Podelski and P. Van Roy. The beauty and beast algorithm: Quasi-linear incremental tests of entailment and disentailment over trees. In M. Bruynooghe, editor, *Proceedings of the 1994 International Logic Programming Symposium*, pages 359–374, Ithaca, 1994. ALP, MIT Press.

[53] E. Pontelli and G. Gupta. Analysis of dependent and-parallelism. In *Proceedings of the fourth Compulog-Net Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Languages*, pages 73–91, September 1996.

[54] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I engine: A parallel implementation of the Basic Andorra Model. Technical note, University of Bristol, Department of Computer Science, March 1990.

[55] V. A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1990.

[56] H. Sato, N. Ichiyoshi, T. Dasai, T. Miyazaki, and A. Takeuchi. A sequential implementation of Concurrent Prolog - based on the deep binding scheme. In *The First National Conference of Japan Society for Software Science and Technology*, pages 299–302, 1984. In Japanese.

[57] C. Schulte and G. Smolka. Encapsulated search in higher-order concurrent constraint programming. In M. Bruynooghe, editor, *Proceedings of the 1994 International Logic Programming Symposium*, pages 505–520, Ithaca, 1994. ALP, MIT Press.

[58] E. Shapiro. *Concurrent Prolog: A Progress Report*, chapter 2, pages 27–83. MIT Press, 1987.

[59] E. Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, 1987.

[60] R. Sharma and M. L. Soffa. Parallel generational garbage collection. In *OOPSLA'91*, pages 16–32, 1991.

[61] K. Shen. Exploiting and-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme DDAS. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 717–731, November 1996.

[62] K. Shen. Overview of DASWAM: Exploitation of dependent and-parallelism. *Journal of Logic Programming*, 29:245–293, October/December 1996.

[63] SICS. *SICStus v3.* URL http://www.sics.se/isl/sicstus.html.

[64] G. Smolka. The definition of Kernel Oz. In A. Podelski, editor, *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910, pages 251–292. Springer, 1995.

[65] P. Stenström and F. Dahlgren. Applications for shared memory multiprocessors. *IEEE Computer*, 29(12), October 1996.

[66] A. Takeuchi and K. Furakawa. *Parallel Logic Programming Languages*, chapter 6, pages 188– 201. MIT Press, 1987.

[67] E. Tick. *Parallel Logic Programming.* MIT Press, 1991.

[68] E. Tick. The devolution of concurrent logic programming languages. *Journal of Logic Programming*, 23(2), May 1995.

[69] H. Ueda and J. Montelius. Dynamic scheduling in an implicit parallel system. In *Ninth International Conference on Parallel and Distributed Computing Systems*, September 1996.

[70] K. Ueda. *Guarded Horn Clauses*, chapter 4, pages 140–156. MIT Press, 1987.

[71] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The computer Journal*, 33(6):494–500, 1990.

[72] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, 1983.

[73] P. R. Woodward. Perspective on supercomputing: Three decades of change. *IEEE Computing*, 29(10), October 1995.

[74] Wm. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

Computing Science Department, Uppsala University

Uppsala Theses in Computing Science

01. Mats Carlsson, *LM-Prolog – the Language and its Implementation*, PhL, 1984
04. Lennart Beckman, *Towards an Operational Semantics for Concurrent Logic Programming Languages*, PhL, 1987
06. Jonas Barklund, *Efficient Logic Data Structures*, PhL, 1988
07. Sven-Olof Nystrm, *On the Semantics of Concurrent Logic Programming Languages: a Variable-free Concurrent Language and Its Operational Semantics*, PhL, 1989
09. Jonas Barklund, *Parallel Unification*, PhD, 1990
10. Håkan Millroth, *Reforming Compilation of Logic Programs*, PhD, 1990
13. Mattias Waldau, *Verification of Logic Programs using Verification Sentences*, PhL, 1990
14. Björn Carlson, *An Approximation Theory for Constraint Logic Programs*, PhL, 1991
16. Johan Bevemyr, *A Recursion Parallel Prolog Engine*, PhL, 1993
17. Margus Veanes, *Cycletrees: a Novel Class of Interconnection Graphs*, PhL, 1993
18. Thomas Lindgren, *The Compilation and Execution of Recursion-Parallel Prolog on Shared Memory Multiprocessors*, PhL, 1993
19. Sverker Janson, *AKL — A Multiparadigm Programming Language*, PhL, 1994
20. Pierangelo Dell'Acqua, *SLD-Resolution with Reflection*, PhL, 1995
21. Björn Carlson, *Compiling and Executing Finite Domain Constraints*, PhD, 1995
23. Magnus Nordin, *IGOR: A tool for developing abstract domains for Prolog analyzers*, PhL, 1995
24. Sven-Olof Nyström, *Denotational Semantics for Asynchronous Concurrent Languages*, PhD, 1996
25. Johan Bevemyr, *Data-parallel Implementation of Prolog*, PhD, 1996
26. Thomas Lindgren, *Compilation Techniques for Prolog*, PhD, 1996

Swedish Institute of Computer Science

SICS Dissertation Series

01. Bogumił Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, PhD, 1990
02. Mats Carlsson, *Design and Implementation of an OR Parallel Prolog Engine*, PhD, 1990
03. Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*, PhD, 1990
04. Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*, PhD, 1991
05. Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*, PhD, 1991
06. Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*, PhD, 1991
07. Roland Karlsson, *A High Performance OR-parallel PROLOG System*, PhD, 1992
08. Erik Hagersten, *Towards Scalable Cache Only Memory Architectures*, PhD, 1992
09. Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*, PhD, 1993
10. Mats Björkman, *Architectures for High Performance Communication*, PhD, 1993
11. Stephen Pink, *Measurement, Implementation and Optimization of Internet Protocols*, PhD, 1993
12. Martin Aronsson, *GCLA: The Design, Use, and Implementation of a Program Development System*, PhD, 1993
13. Christer Samuelsson, *Fast Natural-Language Parsing Using Explanation-Based Learning*, PhD, 1994
14. Sverker Jansson, *AKL—A Multiparadigm Programming Language*, PhD, 1994
15. Fredrik Orava, *On the Formal Analysis of Telecommunication Protocols*, PhD, 1994
16. Torbjörn Keisu, *Tree Constraints*, PhD, 1994
17. Olof Hagsand, *Computer and Communication Support for Interactive Distributed Applications*, PhD, 1995
18. Björn Carlsson, *Compiling and Executing Finite Domain Constraints*, PhD, 1995
19. Per Kreuger, *Computational Issues in Calculi of Partial Inductive Definitions*, PhD, 1995
20. Annika Wærn, *Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction*, PhD, 1996
22. Klas Orsvärn, *Knowledge Modelling with Libraries of Task Decomposition Methods*, PhD, 1996
23. Kristina Höök, *A Glass Box Approach to Adaptive Hypermedia*, PhD, 1996
24. Bengt Ahlgren, *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption*, PhD, 1997

Printed in Stockholm Sweden by Gotab using a camera-ready copy typeset in Computer Modern by the author, using a LaserWriter, the LaTeX document preparation system, the Emacs editor, and a SPARCstation 20 workstation.