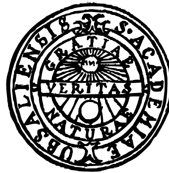
UPPSALA THESES IN COMPUTING SCIENCE
26

Compilation Techniques for Prolog

Thomas Lindgren

Computing Science Department
Uppsala University

Thesis for the Degree of
Doctor of Philosophy

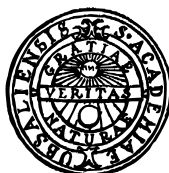


UPPSALA 1996

Compilation Techniques for Prolog

Thomas Lindgren

A Dissertation submitted in partial fulfilment of the requirements for the Degree of Doctor of Philosophy at Computing Science Department, Uppsala University.



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

Uppsala Theses in Computing Science 26
ISSN 0283-359X
ISBN 91-506-1181-X

(Dissertation for the Degree of Doctor of Philosophy in Computing Science presented at Uppsala University in 1996)

Abstract

Lindgren, T. 1996: Compilation Techniques for Prolog, *Uppsala Theses in Computing Science 26.* 36pp. Uppsala. ISSN 0283-359X, ISBN 91-506-1181-X.

Current Prolog compilers are largely limited to optimizing a single predicate at a time. We propose two methods to express the global control of Prolog programs. The first method transforms a Prolog program into a continuation-passing style, where all operations have explicit success and failure continuations. The second method directly constructs a control flow graph from the Prolog program. This is done by first performing an analysis that determines the potential targets of success or failure for every predicate, then by using this information to build the control flow graph. We develop an optimization, factoring, that reduces the size of the analysis solution substantially.

Uninitialized output variables are an important feature of all high-performance implementations of logic programming language today. We show that a polyvariant output variable analysis is more precise and robust than the previously proposed monovariant analyses, while remaining fast and practical.

Garbage collection is a potential bottleneck in a high-performance Prolog implementation. We show that generational copying garbage collection can be adapted to Prolog, and that such a collector is considerably faster than currently used Prolog garbage collectors. A number of problems specific to Prolog and its implementation techniques are presented and solutions provided.

Finally, we propose Reform Prolog, a recursion-parallel dialect of Prolog. We show that recursion-parallelism can be efficiently exploited on today's shared memory multiprocessors by means of a restricted execution model, a streamlined runtime system and sophisticated compilation techniques. Our experiments indicate that Reform Prolog runs programs with low parallelization overheads (compared to standard sequential Prolog implementations), yet attains high speedups even on large multiprocessors.

Thomas Lindgren, Computing Science Department, Uppsala University, Box 311, S-751 05 Uppsala, Sweden.

© Thomas Lindgren 1996.

ISSN 0283-359X
ISBN 91-506-1181-X
Printed by Graphic Systems AB, Stockholm 1996.

Acknowledgments

First of all, thanks to Håkan Millroth, my advisor. His guidance and mentoring have been invaluable.

Johan Bevemyr has been a close friend and colleague ever since our undergraduate days. Without his hard work on Reform Prolog and his knack for low-level implementation, things would have been much harder, not to mention less fun.

Per Mildner has been a great help and a conscientous critic of my later work. I have been fortunate in having ready access to his experience.

Thanks to my external examiner, Peter Van Roy, and to my readers, Jonas Barklund, Mats Carlsson, Seif Haridi and Sven-Olof Nyström. I appreciate your efforts.

The computing science department has provided a stimulating setting for research. In particular, Patric Hedlin, who helped with hacking Reform Prolog; Magnus Nordin, who provided perspectives on static analysis, computer graphics and animation and many other things; and Mattias Waldau, who deserves a big thanks for early inspiration.

I have been fortunate in discussing the ideas of this thesis with many of the premier people in logic programming. In particular, I would like to thank Saumya Debray, Ulrich Neumerkel, Paul Tarau and David H.D. Warren for discussions over the years.

John Lloyd and Tony Bowers were helpful and hospitable during a visit at Bristol; Malcolm Brown kindly helped me and Johan at the Edinburgh Parallel Computing Centre. Thanks to all of you.

Finally, many thanks to my friends and family for making these years enjoyable.

Prior publication

The papers in this thesis have been published previously. They have not been edited, except to adjust typography, remove typographical errors and update references.

A. T. Lindgren, A continuation-passing style for Prolog, in *Proc. Intl. Logic Programming Symposium 1994*, MIT Press, 1994.

B. T. Lindgren, Control flow analysis of Prolog, in *Proc. Intl. Logic Programming Symposium 1995*, MIT Press, 1995.

C. T. Lindgren, Polyvariant detection of uninitialized arguments of Prolog predicates, Journal of Logic Programming. Vol 28(3) Sept. 1996, pp. 217-229.

D. J. Bevemyr, T. Lindgren, Simple and efficient copying garbage collection for Prolog, in *Programming Language Implementation and Logic Programming 1994*, LNCS 844, Springer Verlag, 1994.

E. J. Bevemyr, T. Lindgren, H. Millroth, Exploiting recursion-parallelism in Prolog, in *PARLE-93*, eds. A. Bode, M. Reeve, G. Wolf, LNCS 694, Springer Verlag, 1993.

F. J. Bevemyr, T. Lindgren, H. Millroth, Reform Prolog: The language and its implementation, in *Logic Programming: Proceedings of the Tenth International Conference*, MIT Press, 1993.

G. T. Lindgren, J. Bevemyr, H. Millroth, Compiler Optimizations in Reform Prolog: Experiments on the KSR-1 Multiprocessor, in *Proc. EURO-PAR'95*, LNCS 966, Springer Verlag, 1995.

The articles are reprinted with the permissions of the publishers.

Contents

1 Summary 1
1.1 Introduction 1
1.2 Summary 5
1.3 Related work 15
1.4 Conclusion 24

2 A Continuation-Passing Style for Prolog 25
2.1 Introduction 25
2.2 Related work 26
2.3 Translation to continuation passing style 28
2.4 Optimizations 33
2.5 Conclusion 37

3 Control flow analysis of Prolog 39
3.1 Introduction 39
3.2 Preliminaries 40
3.3 Making control explicit 41
3.4 Control flow analysis 43
3.5 Constructing the CFG 47
3.6 Evaluation 48
3.7 Related work 51
3.8 Conclusion 52
3.9 Appendix: Example of control flow constraints 52
3.10 Appendix: classes of strongly connected components 53

4 Polyvariant Detection of Uninitialized Arguments of Prolog Predicates 55
4.1 Introduction 55
4.2 Transforming predicates 56
4.3 Transforming programs 58
4.4 Examples 60
4.5 Empirical evaluation 63

4.6 Related work . . . 68
4.7 Conclusion . . . 70

5 A Simple and Efficient Copying Garbage Collector for Prolog 71
5.1 Introduction . . . 71
5.2 Related work . . . 72
5.3 Algorithm . . . 74
5.4 Introducing generational garbage collection . . . 78
5.5 Evaluation . . . 80
5.6 Conclusion . . . 83
5.7 Acknowledgment . . . 83
5.8 Appendix: new reference chains may appear from variable shunting . . . 84

6 Exploiting Recursion-Parallelism in Prolog 85
6.1 Introduction . . . 85
6.2 Reform Prolog . . . 86
6.3 Recursion parallel programming . . . 88
6.4 The parallel abstract machine: overview . . . 90
6.5 Compiling recursion parallel programs . . . 91
6.6 Instruction set . . . 92
6.7 Experimental results . . . 93
6.8 Conclusions . . . 97
6.9 Appendix: benchmark plots . . . 98

7 Reform Prolog: The Language and its Implementation 101
7.1 Introduction . . . 101
7.2 Reform Prolog . . . 102
7.3 The parallel abstract machine . . . 104
7.4 Data sharing . . . 106
7.5 Process management . . . 107
7.6 Compiling recursion parallel programs . . . 110
7.7 Instruction set . . . 110
7.8 Experimental results . . . 116
7.9 Conclusions . . . 119

8 Compiler Optimizations in Reform Prolog: Experiments on the KSR-1 Multiprocessor 121
8.1 Introduction . . . 122
8.2 Reform Prolog . . . 122
8.3 Compiler analyses . . . 124
8.4 Compiler optimizations . . . 126
8.5 Experimental setup . . . 127
8.6 Compiler performance . . . 128

8.7 Analysis results 128
8.8 Performance of compiled programs 129
8.9 Effectiveness of optimizations 130
8.10 Conclusion 132
8.11 Appendix A: analysis measurement data 133

Bibliography 135

A Variable classification 149

B Code Generator Internals 151
B.1 Code generator structure 151
B.2 Code generation for recursion-parallel predicates 151
B.3 Code generation for parallel predicates 152

C Performance of Reform Prolog 157
C.1 Reform Prolog vs. SICStus Prolog 157
C.2 Reform Prolog vs. other parallel systems 158

D Parallel Overheads of Reform Prolog 161
D.1 Summary and discussion 168
D.2 Space overheads 169

Chapter 1

Summary

In this chapter, we briefly introduce the logic programming language Prolog, the compilation of Prolog and parallel execution of Prolog programs. We then summarize the rest of the chapters of this thesis, present the related work in the field, and summarize our contributions.

1.1 INTRODUCTION

Prolog

Prolog is a programming language based on a subset of predicate logic where programs consist of Horn clauses. Robinson [114] showed that resolution could be used to efficiently prove theorems for such theories; Colmerauer [37] and Kowalski [78] subsequently noted that such clauses could be viewed as *programs* in addition to logical theories.

For example, consider a predicate specifying list concatenation such that the third argument is the concatenation of the first two arguments. We can write this as two Horn clauses, as shown below.

```
concatenate([],Ys,Ys).
concatenate([X|Xs],Ys,[X|Zs]) :- concatenate(Xs,Ys,Zs).
```

Concatenate/3 is a simple inductive definition. If the first argument is the empty list, then the concatenation of the first two arguments is simply the second argument. If the first argument is a list [X | Xs], with head X and tail Xs, and the second argument is Ys, then the third argument is the concatenation if Zs is the concatenation of Xs and Ys, and the third argument is [X | Zs].

Consider the query:

```
?- concatenate([1,2,3],[4,5],A).
```

It is easy to prove that A = [1,2,3,4,5] is a solution.

However, logic programming differs from functional programming in that predicates can be used in several modes. Consider the following query.

```
?- concatenate(A,B,[1,2]).
```

This query has several solutions. The Prolog system will attempt to find one of them, and then produce more solutions on demand. Prolog employs an incomplete search and control strategy, which may fail to terminate even if solutions can be found in finite time. The use of such a strategy separates Prolog from pure theorem proving.

Many good books on the basics of Prolog and logic programming have been published, such as Sterling and Shapiro's textbook [137] on Prolog programming, or Lloyd's text on logic programming theory [89]. The novice reader is directed to those publications; we will in the rest of this thesis assume a working knowledge of the concepts and programming methods of Prolog and logic programming, such as resolution, unification, backtracking, cut, clause indexing, higher order predicates, and so on.

Compilation of sequential Prolog

A Prolog compiler maps resolution and unification into executable code. Since the distance between the two is considerable, the first implementations were interpreters. However, Warren [158] developed a compiled implementation of Prolog in the late 1970's. In 1983, he proposed his 'new engine', or Warren's abstract machine (WAM), as it became known [159].

The WAM inspired considerable research in Prolog implementation. The main goals of WAM were to optimize control by clause indexing and environment trimming, and to compile unifications into efficient low-level code. A previous design [158] also provided some of these concepts in a different setting. Several commercial and academic designs have been built on the WAM, including Quintus Prolog, SICStus Prolog and Eclipse.

Warren's design has been the starting point of two major directions of research. The first of these is to improve on Warren's engine by local optimizations which target each clause or predicate in isolation [26]. An example of this is when clauses fail early. In this case, backtracking can be optimized [27, 96]. A second example is that unifications can be compiled into very efficient code [150, 91, 95]. As a third example, clause indexing can be improved by sophisticated source-to-source transformation or code generation [42, 29].

The drawback of local techniques is that they cannot take advantage of the calling context. For example, it may be the case that a predicate is always

called in a single mode. Taking advantage of this information can significantly improve clause indexing and the compilation of primitive operations, by reducing both code size and execution time.

The second approach is to perform a global dataflow analysis of the program, and then annotate each predicate and/or goal with the facts holding when the predicate (goal) is reached. Among the dataflow facts detected by current compilers are the following.

- The mode of a variable.
- The possible bindings of a variable.
- Whether a variable needs to be dereferenced or trailed.
- Whether variables are aliased.

The analysis information can be used to improve clause indexing (e.g., by deleting type tests that are statically determined) or by improving unifications (e.g., if the involved variables are bound or free, or may be bound efficiently) [154, 142]. Compilation based on global analysis yields good or excellent speedups over systems lacking analysis [155, 143]. See also Van Roy's excellent survey of the implementation work for Prolog and logic programming [156].

Compilation of parallel Prolog

As multiprocessor computers become increasingly common, another means of reducing execution times presents itself: distributing the computation on several processors and running them in parallel.

Exploiting parallelism in Prolog is conceptually easy enough: perform multiple resolution steps in parallel. If a goal is resolved with several clauses at once, we say that the system extracts *or-parallelism*. If several goals are resolved simultaneously, the system extracts *and-parallelism*. When the resolved goals share free variables, the parallelism is classified as *dependent and-parallelism*. When goals do not share free variables, the parallelism is said to be *independent and-parallelism*.

Or-parallel systems have been successfully built for some years [2, 90] and have been applied to search problems. In recent years, or-parallel constraint solving has emerged as a potential application area [31, 147].

For and-parallel systems, the main problem is the management of parallel backtracking. An and-parallel system must handle process failure, which may require notifying consumers of the bindings the failing process has produced, as well as telling some other process to perform backtracking.

A secondary problem is to ensure that sequential and parallel execution produce identical answers (i.e., yield the same solutions and side-effects in the same order).

There are several solutions to these problems. The first, and perhaps simplest, is to define a new language, possibly better suited to concurrent execution. This approach was taken for languages such as Concurrent Prolog [126], Parlog [34] and GHC [152], and was termed stream and-parallelism: processes are goals, and communicate by means of shared variables that implement, e.g., streams of messages. Tick [146] gives a good overview of the state of the art in this field. An example of a recent implementation of such a language is the Monaco compiler [145].

A second approach, compatible with Prolog, is to extract independent and-parallelism [49, 67]. A failed process needs in this case only notify its siblings of the failure, which is substantially simpler than full dependence checking. A number of systems have been built to exploit independent and-parallelism [68, 63]. Subsequently, the notion of independence has been refined so that processes may share variables but still are independent as long as they do not affect each other's control [69].

The third approach (again used with Prolog) is to only bind variables when the binding process is deterministic. No dependence checking is then needed, because the producer of the binding cannot backtrack to produce another binding [162, 160, 103]. This is also called *determinism-driven and-parallelism*, and has been the basis of a number of research systems.

A fourth approach is to record the dependences between processes dynamically, so that backtracking can be done properly. This approach is taken by Conery and Kibler [38], and subsequently by Shen [127, 128, 129]. Dependence recording and dependence checking delegated entirely to runtime is too inefficient. However, Shen [127] has shown that by carefully selecting the dependences to record at compile-time, and otherwise run only independent goals in parallel, one can construct a system competitive with less general dependent and-parallel systems such as Andorra-I [163].

A fifth, and fairly recent, approach is to exploit *data-parallelism*, where a single program operates simultaneously on a (preferrably large) collection of data. The *single-program, multiple-data* (SPMD) paradigm, where each process is a parallel loop iteration, exploits data-parallelism and is arguably the most popular programming model for conventional multiprocessors. In logic programming, data-parallelism has been used in the contexts of or-parallelism [132, 147] and and-parallelism [11, 100, 12, 157].

The sixth and final approach combines the previous methods. A program may lack parallelism of a given form, which will yield poor performance on systems that extract only that form of parallelism. Some researchers

have set the goal to extract *maximal parallelism* from Prolog programs, by simultaneously exploiting multiple forms of parallelism [60, 63, 62, 61, 118].

1.2 SUMMARY

This section summarizes the papers in this thesis and discusses the scientific contributions, and my personal contributions.

In Prolog programs, the clause and goal selection rules are implicit. Paper A shows how to translate Prolog into a form where control is explicit, which simplifies the underlying implementation. Paper B shows how to derive a control flow graph from a Prolog program. Debray [44] has shown that the control flow graph is a useful tool for an optimizing compiler. Paper C proposes a robust static analysis to detect uninitialized arguments to predicates; this information can be used for important compiler optimizations. Paper D shows how to adapt generational copying garbage collection to Prolog, and demonstrates that this has advantages over previous mark-sweep algorithms. Papers E, F and G develop the design and implementation issues of Reform Prolog, a data-parallel dialect of Prolog.

Continuation-passing style for Prolog

A fundamental problem in executing Prolog is how to efficiently map Prolog's control (including recursion and backtracking) onto conventional hardware.

The traditional solution to this problem is to translate the Prolog code into abstract machine instructions that manage the control structures of Prolog, typically using a variant of WAM. The disadvantage with this approach is that the WAM is quite complex [159, 1] and thus difficult to implement, optimize and reason about.

While most Prolog implementations use a Prolog-to-WAM compiler, an alternative approach, *binarization*, has been proposed by Tarau [139, 140, 141] and Demoen and Mariën [51, 50]. The idea is to manage control by transforming general Prolog programs into a subset of Prolog that can be implemented more easily. When binarization is applied globally (i.e., for the entire program), the result is that every clause has a single goal. This removes the need for a goal selection rule in the process of resolving a clause. The advantage is that the underlying abstract machine is considerably simpler than WAM while being competitive in performance [140].

Binarization does not compile the clause selection (search) rule, however, but relies on WAM's scheme for backtracking. Paper A shows how to express failure control on the source level, just as binarization expresses success control. The idea is to rewrite the program so that every predicate consists

of a single, binary clause. Resolution of such a clause is trivial: replace the head of the clause with the body. All static control decisions of the program have thus been compiled away.

We develop a translation of Prolog into *binary continuation style* (BCS): each predicate takes two extra arguments, being the success and failure continuations of the predicate. First rewrite all predicates into *indexed single-clausal form*, where each predicate collapses its clauses into a single clause with explicit indexing operations (using, e.g., if-then-else and disjunctions). Each new predicate is then rewritten into a collection of BCS predicates and a number of metacall clauses that handle failure and success control when predicates return.

The generated programs have the following properties.

- Primitive operations P have explicit success and failure continuations.
- Nondeterministic choices are coded by (a) saving the current state, (b) building a continuation that restores the saved state, and (c) passing this as a failure continuation.
- All user-defined 'entry point' predicates are defined by a single, binary clause. They take two extra arguments: a success continuation and a failure continuation. These extra arguments can be viewed as 'labels with arguments' in the sense of the original continuation-passing style [136].
- The metacall clauses can be viewed as implementing two jump tables, or as code pointers on the implementation level.

The paper concludes with a brief discussion on some possible optimizations. First, the representation of environments and choicepoints is explicit, and thus open to compiler optimization. For example, a compiler may arrange so that environments share data, or so that environments and choicepoints share data. Second, the compiler can share code between predicates, in a spectrum of options ranging from WAM to binarization. Third, since state save/restore operations are explicit, the compiler can eliminate them or move them around.

Scientific contributions The paper extends binarization to encompass failure control. The new intermediate format, binary continuation style, can be used by an optimizing compiler.

Control flow analysis

Paper B discusses how to construct a control flow graph (CFG) for a Prolog program. The difficulty lies in properly accounting for tail recursion, backtracking and cuts.

The main problem is to determine where control passes when a predicate succeeds or fails. The solution chosen in the paper is to form a collection of set constraints. The analyzer first annotates the program with labels, indicating targets of success or failure. It then walks all the predicates of the program and formulates local constraints, e.g., on the form "if predicate p fails, it may backtrack into predicate q".

We solve these constraints by transforming them into a directed graph and reducing the graph with Tarjan's strongly connected components algorithm [122] into a directed acyclic graph. The reduced graph is then traversed to find the labels reachable by each node.

Given the control flow solution, we show how to construct a control flow graph (CFG) for the program. Our measurements indicate that success control is unproblematic, while failure control is less straightforward. However, in some smaller benchmarks, all backtracking could be statically resolved as jumps.

The size of the control flow solution (a set of labels per variable) may be impractical for large, nondeterministic programs. We show that the reduced graph has considerable structure which can be exploited to share the solution between variables, which we call *factoring*. For 11 out of 26 benchmark programs, the factored solution is half the size of the original solution. For a further seven programs, the original solution is 50% larger than the factored solution.

Scientific contributions The paper is the first to show how to construct a global control flow graph (encompassing the entire program) from a Prolog program, using a simple and efficient algorithm based on set constraints. It also shows how to reduce the size of the control flow solution by factoring the solution and sharing it between predicates.

Measurements on a set of standard benchmarks indicate that success control is typically simple, while failure control is more complex and sometimes very complex.

Polyvariant detection of uninitialized arguments

In practice, many Prolog predicates have arguments that are always unbound and unaliased. These *uninitialized variables* serve as output arguments, locations where the predicate returns information. This is good

news for a Prolog compiler. An uninitialized variable need not be trailed or dereferenced when it is bound, nor need we initialize its memory location beforehand. Optimizing uninitialized variables can lead to considerable execution time savings, and is used by all high-performance logic programming implementations today [155, 142, 133, 59].

While some systems require the user to declare output arguments, Prolog compilers traditionally derive such information by global analysis. A simple and effective approach is taken by Van Roy [155] and Getzinger [57]. However, their analysis has the drawback of being *monovariant*: all calls to a predicate in the entire program are used to summarize which arguments are uninitialized. The drawback is that a seldom-visited call site may be summarized with a frequently-visited call site and ruin valuable information at that site, which leads to less aggressive optimization. In essence, the monovariant analysis is brittle.

Paper C proposes the use of a *polyvariant* algorithm instead: multiple versions of a predicate may be generated, depending on the available information. The approach taken in the paper is to generate a new predicate version for every combination of initialized/uninitialized arguments.

Some questions are raised by this approach. First, the polyvariant algorithm may generate a number of predicate versions exponential in the number of arguments. Is the analysis practical? The paper shows that over a large number of benchmarks, ranging from tens to several thousands of lines of code, the number of versions per predicate is moderate: the number of predicates increases by approximately 20% over the original when program entrypoints are declared.

Second, does the polyvariant analysis yield better results than the monovariant analysis? Our measurements show that for fifteen programs, the monovariant and polyvariant algorithms yield the same results, while for sixteen programs, 2-26% of the predicate arguments are conditionally uninitialized (i.e., uninitialized at some but not all call sites). In three programs, two of which are 2000 lines of code or more, the monovariant analysis does not find *any* uninitialized arguments, while the polyvariant algorithm still detects 17-26% uninitialized arguments. The monovariant algorithm naturally enough never provided more precision than the polyvariant algorithm. Generally, the gap between the monovariant and polyvariant algorithm increased when we looked at only the large programs.

Finally, can the analysis be completed in reasonable time? In general, the polyvariant algorithm required less than a second to run on a 55-MHz workstation with SICStus native code. Even the largest program required only 4.5 seconds. Hence, such an analysis does not appear to be a bottleneck in a compiler.

Scientific contributions The paper shows that a polyvariant algorithm for uninitialized arguments is fast, practical (does not result in code explosion), and robust (always as precise as a monovariant algorithm and retains precision even in difficult cases).

Generational garbage collection

Paper D consider the problem of garbage collection for Prolog. In particular, the paper shows that copying and generational copying garbage collection can be implemented straightforwardly and efficiently on standard hardware.

The basic idea of the paper is to adapt an efficient collector, such as Cheney's algorithm [30]. The paper solves a number of problems that appear when adapting generational copying collection to Prolog.

Interior pointers are common in Prolog. An interior pointer is one that points directly to a field in a structure, without referring to the head of the structure. This implies that only parts of a structure may be live when a collection occurs, but also that there is no way to tell directly whether a pointer is interior or not.

We say that the *live data* at a given point is the data reachable by traversal starting from a set of root registers. A copying collector migrates the live data from an old area, called from-space, into a new area, called to-space.

Ideally, the algorithm should only migrate live data into to-space. If a structure has live subfields but is itself dead, then only the live subfields should be migrated. On the other hand, if the entire structure is live, then the entire structure must be copied as a unit. This is a conundrum for a straightforward copying algorithm, since it may encounter pointers to the subfields before discovering any pointers to the structure itself. Simply copying the subfield makes subsequent copying of the entire structure very difficult: the already copied fields must somehow be moved back inside the structure and pointers to the fields must be adjusted.

Paper D solves the problem of interior pointers by using a mark-copy algorithm. First, the live data in from-space are marked by traversing the data reachable from a set of roots. Then, the marked data are copied into to-space.

The next problem with copying collection is that the ordering of heap data is not retained after a collection. Consider the Prolog heap as a stack of segments delimited by choicepoints. On backtracking, a Prolog implementation can reclaim the topmost segment, since all data therein belong to a failed branch of the proof tree. This has led to the general use of mark-sweep algorithms that carefully preserve the segments. However, Cheney-style

copying [30] does not preserve the segment ordering of data. Instead, it traverses the live data in breadth-first order.

There are three problems with not preserving the ordering of data on the heap. First, memory can not always be reclaimed on backtracking. Second, Prolog implementations record conditional bindings on a trail stack, and can avoid recording a binding ("trailing the binding") in reasonably common situations. However, for our algorithm, bindings in the copied area must always be trailed (rather than occasionally). Third, variables may have been compared by special Prolog primitives, e.g., `@</2`. This is generally implemented by address comparison, and requires the relative ordering of variables to be maintained.

The proposed collector exploits that data allocated since the last collection still retain the desired heap ordering. Hence, memory allocated after the last collection can still be reclaimed by backtracking. Measurements show that the copying algorithm in practice recovers as much memory on backtracking as a conventional ("perfect") mark-sweep algorithm on a range of realistic benchmarks. Furthermore, the extra amount of trailing is shown to be negligible in practice: never more than one-quarter of a percent of the total number of trailings.

The proposed solution to the problem of retaining the order of compared variables is to migrate unbound variables to a dedicated stack at the time of comparison. That stack is then collected using a mark-sweep algorithm to retain the ordering between variables, while copying is used for the rest of the heap.

Finally, the paper shows how to extend the copying algorithm to generational collection. Generational garbage collection [84, 4] relies on the observation that newly created objects tend to be short-lived. Thus, garbage collection should concentrate on recently created data. The heap is split into two or more generations, and the most recent generation is collected most frequently. When the youngest generation fills up, a collection spanning more generations is done, and the survivors move to the oldest of these generations. The main difficulty is to record pointers from older to newer generations, which is needed to identify all live data. This is also called a *write-barrier*, since writing the old generation requires some checking and possibly recording the written pointer.

The crucial insight is that pointers from the old generation to the new generation (in a two-generation system) can be found by scanning the trail stack. By changing the trailing mechanism so that all bindings in the old generation are trailed, we get a write-barrier almost for free. The only extra cost is some unnecessary trailings in some situations. This cost is again negligible for the measured benchmarks.

Scientific contributions The paper shows how conventional garbage collection techniques can be adapted to Prolog by solving a number of problems specific to Prolog: interior pointers, heap ordering of data and write-barriers for generational garbage collection.

Measurements indicate that a number of potential problems with mark-copy and generational mark-copy are unproblematic in practice, that mark-copy beats mark-sweep, and that generational mark-copy beats mark-copy.

Author's contributions The construction of the garbage collection algorithms and experimental evaluation in the paper were done in close collaboration with Johan Bevemyr. Bevemyr wrote the actual collector code, collected the benchmarking data and integrated the three collectors with Reform Prolog.

Reform Prolog

Papers E and F introduced Reform Prolog, a language built on Millroth's principle of recursion parallelism [97]. Paper G evaluates compiler optimizations that reduce the number of synchronizations and locking operations needed during parallel execution.

Consider the list recursive predicate p.

$$\begin{array}{lcl} p([], \ldots) & \leftarrow & \Upsilon \\ p([X|Xs], \ldots) & \leftarrow & \Phi \wedge p(Xs, \ldots) \wedge \Psi \end{array}$$

Millroth [97, 100] showed that such predicates could be compiled into efficient loop code, using *Reform compilation.*

1. Compute the length of the input list as n and build the n instances of Φ and Ψ.
2. For i ranging from 1 to n, execute Φ_i.
3. Execute the remaining recursive call to p.
4. For i ranging from n to 1, execute Ψ_i.

Millroth also showed that nonlinear recursion could be compiled into loop code [99]; this is beyond the scope of our work.

The Reform Prolog project was started in September 1990 to study the parallelization of Prolog programs by means of Reform compilation. Originally,

the project members were Håkan Millroth, Margus Veanes and the author. Johan Bevemyr joined the group in late 1990. It was decided that dependent and-parallelism, where goals communicate by shared logic variables, was the most interesting venue.

An early prototype design, reminiscent of Shen's DDAS [127] in that it handled 'roll-backs' between processes (i.e., resetting the readers of a variable binding if the producer of the variable binding failed and removed the binding), ran on a distributed memory machine at Edinburgh's Parallel Computing Centre in September 1991. Our experiences with this system were mainly negative: tracking dependences between processes was quite expensive, and the combination of distributed memory (with explicit process communication) and pointer datastructures required considerable marshalling and unmarshalling when communication occurred. In 1992, Bevemyr and the author formulated a second design with the intent to optimize or remove, rather than tolerate, the difficult features of implementing a parallel Prolog system. Shen [129] had shown that automatic parallelization of sequential logic programs did not yield the huge speedups required to efficiently use massively parallel machines; thus, we instead put the responsibility of extracting parallelism on the programmer, and defined a simple language and execution model that could be implemented very efficiently.

Rather than trying to extract maximal parallelism from an arbitrary Prolog program, Reform Prolog parallelized a restricted and well-defined class of programs. Rather than accepting backtracking between processes, Reform Prolog enforced binding determinism and process determinism by a mixture of compile time and runtime means. Rather than using a sophisticated runtime system to control the parallel computation and exploit local opportunities for parallel execution, Reform Prolog only extracted parallelism from user-declared recursion-parallel predicates. Finally, rather than aiming for large speedups relative to a parallel system running on one processor, the Reform Prolog system was designed for high *absolute* speedups, with respect to a sequential system. This was done by extending a sequential WAM and confining overheads due to parallelism to process interactions (locking and suspension). It would then be straightforward to use available sequential compiler technology on the sequential portions of the code, should there be a need. Some encouraging initial experiments with the new system were run and reported at a JICSLP'92 workshop.

Execution model The Reform Prolog execution model gives programmers a model for parallel execution and parallel performance. It furthermore specifies how time-dependent operations, such as var/1 or indexing, are managed.

1. Binding a variable X that may be tested by time-dependent operations can only be done when the binding goal is leftmost in the resolvent. Testing the value of X must delay until X is bound or the loop iteration is leftmost.

2. Bindings to variables shared between loop iterations may never be conditional [103].

3. When a loop iteration is finished, it is always deterministic.

4. A (parallel) loop iteration can not start a parallel loop.

Further details, e.g., on handling predicates with side-effects, can be found in Refs. [17, 85].

By making the first restriction, we ensure that parallel and sequential execution yield the same answer. By making the second and third restrictions, we can guarantee that no costly 'rollback-on-failure' operations need to be supported. (See for instance Shen's DDAS for a design where such rollbacks are supported [127, 129].) The fourth restriction means the runtime system can be kept simple and efficient. All processors share a single parallel loop, which makes scheduling and leftmostness checking straightforward.

Compiler optimizations Reform Prolog puts the responsibility of keeping track of time-dependent and shared variables on the compiler. The Reform Prolog compiler must emit explicit locking operations when shared or time-dependent variables are bound, and preserves sequential semantics by emitting suspension instructions when time-dependent variables may be bound. This means the compiler can also *avoid* such instructions when it can determine that they are not needed. In fact, the compiler emits ordinary WAM code when the involved data are known to be local to the process.

In order to conservatively approximate which variables are shared or time-dependent, the compiler employs a static analysis. The static analysis determines modes, aliasing and linearity of variables. For parallel code, the analyzer also tracks whether variables *may be* robust or fragile: by tagging variables shared between loop iterations as robust when the loop is begun, and by tagging robust variables as fragile when they are tested. Finally, the analyzer also keeps track of whether loop iterations are deterministic or not, at every goal and at the end of an iteration.

Code generation using this information is described in greater detail in the author's licentiate thesis [85]. Briefly, the compiler uses knowledge of locality, modes and determinism to optimize operations. Many primitive operations permit synchronization and locking operations to be omitted when data are known to be instantiated or local.

Experimental results For a 24-processor Sequent Symmetry, we found parallelization overheads on the order of 2-12% as compared to a sequential Prolog system. Competing systems such as Andorra-I and NUA-Prolog show parallelization overheads of 35-40% and 50%, respectively (for 10 and 11 processors, respectively). For a 48-processor KSR-1, we found parallelization overheads of 0-17%, with larger benchmarks on the order of 2-6% .

Finally, two compiler optimizations were evaluated: removing suspensions and removing locking instructions. We found that optimizing suspensions was vital to the complex benchmarks, since they otherwise have to suspend almost immediately (although an interesting exception to this rule was found). Optimizing lockings was less successful. Even though the compiler reduced the number of executed lockings to 52% of the unoptimized number, there was no change in execution times. We attribute this to the infrequency of locking operations, and to the execution model of Reform Prolog.

Scientific contributions Papers E, F and G contribute a number of insights to the design and implementation of and-parallel logic programming systems.

- Data-parallelism is a rich source of parallel work even for logic programs, and recursion-parallelism is an appropriate tool to exploit such work.
- Reform Prolog confirms Naish's insight that binding determinism [103] is a crucial implementation concept for dependent and-parallel systems.
- Static analysis can accurately separate shared data from local data, which reduces the need for locking instructions (e.g., reduce the dynamic number of lockings to 52% of the unoptimized total).
- Static analysis can accurately identify the data accesses that sequentialize the parallel workers, and the compiler can avoid unnecessary sequentialization using this information.

Reform Prolog showed that a simple data-parallel Prolog implementation could attain significantly greater speedup and efficiency than previous comparable parallel logic programming systems.

Author's contributions Reform Prolog's execution model was designed in close collaboration with Johan Bevemyr and Håkan Millroth. The Reform Prolog compiler was written by me, while the execution engine was written by Bevemyr. The compiler optimizations were designed in collaboration with with Johan Bevemyr and implemented by me.

Reflections and later work Our results inspired a number of subsequent papers [70, 112, 111] which incorporate some of the Reform Prolog techniques into control-parallel systems, in particular the 'fast unfolding' of Reform compilation. Examples of such systems are &-Prolog [68] and ACE [63]. Results have generally been encouraging.

Recently, Bevemyr [18] and the author [88] have proposed methods to extend the 'flat' Reform Prolog system into efficiently executing arbitrarily nested recursion-parallel loops.

1.3 RELATED WORK

Paper A

There has been considerable interest in transformations of Prolog into Scheme and partial continuation passing styles and the relation of Prolog to intermediate representations, such as Warren's abstract machine [159]. We review earlier research in this section.

Warren's abstract machine It is interesting to compare our intermediate format with Warren's abstract machine, WAM [159].

- Our notation extends the scope of compilation (the 'chunks' or 'basic blocks') by considering actions taken on failure.
- Indexing can be expressed as a source-to-source transformation and also extended to utilize other mutually exclusive tests to restrict the number of candidate clauses.
- We can trade code size against possible unnecessary data allocation by intelligent generation of continuation clauses. Our formulation admits both the approach used in WAM, that of simple binarization, as well as a spectrum of trade offs in between.
- Continuation representation is explicit, thus allowing trimming of choice points as well as environments. Our notation represents choicepoints and environments as terms, which means a compiler can change the parameter passing conventions to suit particular situations, alter the representation of choicepoints and enviroments, and so on. In the WAM, choicepoints and environments have a fixed structure which is inaccessible to the compiler.
- Internal registers are saved separately from argument registers, which implements shallow backtracking by default rather than as a special optimization. We can also admit shallow backtracking in several levels by nested saving of internal registers.

On a more fundamental level, the WAM hides much of the control flow of the computation from the compiler writer. This simplifies compilation but excludes some optimizations. Our representation directly admits a translation into control flow graphs, since control flow is explicit.

Translations in Prolog Binarization, as proposed by Tarau [139] and Demoen and Mariën [51, 50] ignores the failure continuation that relates the different clauses of a predicate. The effects of this is that cuts are handled ad hoc, that control constructs other than conjunction must be handled by rewriting the program so that only conjunctions remain, and that an implicit control stack remains (the stack of choicepoints). Our binary continuation style avoids these problems and also allows us to directly describe indexing as a source-to-source transformation, which binarization delegates to the abstract machine.

Nilsson [106] shows how to derive the WAM choicepoint instructions by partial evaluation of a meta interpreter. Our work is distinct from the control aspects of the WAM and adds translations of other control constructs such as cut.

Brisset and Ridoux [24] propose a CPS for λProlog. As that language has λ-abstractions, their translation is similar to Scheme or Lisp translations discussed below, but does not explicitly manage substitutions. We have found that explicit operations that save and restore state can be useful, since they then can be moved and removed by program transformations. Since they generate higher-order terms in their translation, a subsequent pass of closure conversion [5] converts the term to a first-order representation. Our algorithm, in contrast, directly generates first-order terms as continuations.

Neumerkel [105, 104] has proposed Continuation Prolog, which allows compilers to manipulate continuations and remove some auxilliary output variables. The new program is then translated to binary or standard Prolog. The transformation is manual in nature, but could possibly be automated.

Optimization of Prolog using control flow graphs Debray [44] proposes several optimizations based on Prolog predicates translated into control flow graphs with success and failure edges. These edges do not correspond to the actual control of the program; for instance, if B_1 continues with B_2 on success, and B_2 on failure jumps to B_3, a success edge is added from B_1 to B_3 even though B_1 will never directly jump to B_3. This framework is then used to formulate several optimizations, such as moving tag tests and dereferencing operations out of loops. We believe that, mutatis mutandis, these optimizations could be carried out on our BCS programs as well.

Translations into Scheme and Lisp Several translations of Prolog into Lisp have been proposed. We take the work of Kahn and Carlsson as representa-

tive of this approach. Kahn and Carlsson [75, 25] propose the use of *upward failure* or *downward success* continuations. The former relies on using lazy streams to produce solutions, while the latter performs a procedure return on failure. We note that both methods to some extent bury the symmetry of success and failure continuations, and that neither method eliminates all control stacks. Kahn and Carlsson [76] then employ partial evaluation to produce quite good Lisp code for naive reverse with modes.

Research at Indiana University showed that Horn clauses could be translated into Scheme by fairly straightforward means [55, 77] and that some care was needed to extend Prolog with continuations [66]. In particular, a generalized trail was required to suspend and resume a proof tree branch. Such operations are beyond the scope of this paper; all our failure continuations obey a stack-like discipline.

Paper B

Debray [44] and Sehr [123, 124] use control flow graphs to optimize sequential programs and extract parallelism, respectively. Both authors considered only intraprocedural control flow. In our formulation, procedure calls disappear in an interprocedural sea of assignments, continuation creation and primitive operations.

Debray and Proebsting [48] have recently shown that control flow analyses can be transformed into parsing problems, and use LR(0) and LR(1)-items to perform the analysis. They consider languages with tail recursion, but lacking backtracking.

Shivers [130] and Jones, Gomard and Sestoft [74] proposed control flow analysis for the purpose of recovering the control flow graph of higher-order functional programs. We have studied control flow analysis for a language with less general and somewhat different control structures, and have shown that the solution can be found quickly and represented compactly.

Paper C

Beer [14] was the first to exploit uninitialized variables. This was done by a runtime approach, where registers and heap cells were tagged as uninitialized when created and modified during execution when unifications and similar operations occurred. He found that a large portion of the dynamically occurring variables actually were uninitialized. On a set of benchmarks, he found that dereferencing and trailing could be substantially reduced.

Van Roy [154] defined a static analysis that (among other things) derived uninitialized arguments. Our monovariant transformation in essence mimics

the 'uninitializedness' subset of Van Roy's analyzer and achieves approximately the same precision [155]. Our work thus shows that this part of Van Roy's analysis can be factored out of the rest of his analysis without losing precision, and that polyvariance further improves the results while avoiding code explosion. Van Roy also developed an algorithm that could return approximately 1/3 of the outputs in registers.

Getzinger improved on Van Roy's analysis and explored some alternatives, but remained within the monovariant framework [57, 58].

Taylor [142] subsequently incorporated uninitialized variables into his Parma compiler, and reported substantial performance gains. No indication of the number of uninitialized arguments derived was given.

The recently developed strongly-typed, strongly-moded logic programming language Mercury [133] restricts programs so that outputs are always uninitialized. A recent release performs a mode analysis similar to the one proposed in Paper C.

Bigot, Gudeman and Debray [22] have developed an alternative to Van Roy's algorithm, which they use to decide which output arguments should be returned in registers, and which should be returned in memory. It may be interesting to consider this for our benchmark set, and to possibly use multiple versions of a predicate for different call sites. The Bigot-Gudeman-Debray algorithm uses a single version per predicate.

Gudeman, De Bosschere, Debray and Kannan [23] have defined call forwarding as a way to hoist type tests out of loops or in general to elide type tests when the call site can statically decide tests in the callee. As shown in the nreverse example, our polyvariant transformation occasionally generates crude 'call forwarding' by breaking out calls with uninitialized arguments. This suggests that we could possibly share code between predicate versions. Van Roy's compiler [154] merges multiple calls that have produced the same intermediate code, which can be seen as the reverse of call forwarding.

We are only aware of two implementations of multiple specialization for Prolog programs: Sahlin's partial evaluator Mixtus [117] can generate multiple versions of a predicate, and Puebla and Hermenegildo [113] have recently used multiple specialization to improve the parallelization of &-Prolog.

We finally note that the proposed transformation can be seen as an abstract interpretation [41] followed by a program transformation based on the derived results.

Paper D

Prolog implementations such as SICStus Prolog use a mark-sweep algorithm that first marks the live data, then compacts the heap. We take the

implementation of Appleby et al. [8] as typical. This algorithm works in four steps and is based on the Deutsch-Schorr-Waite [121, 36] algorithm for marking and on Morris' algorithm [101, 36] for compaction (a more extensive summary is given in Paper D). Our copying collector uses a similar mark-phase, but copies data rather than compacting them.

Touati and Hama [149] developed a generational copying garbage collector. The heap is split into an old and a new generation. Their algorithm uses copying when the new generation consists of the topmost heap segment, i.e., no choice point is present in the new generation, and no troublesome primitives have been used (primitives that rely on a fixed heap ordering of variables). For the older generation they use a mark-sweep algorithm. The technique is similar to that described by Barklund and Millroth [13] and later by Older and Rummell [107].

We show in Paper D how a simpler copying collector can be implemented, how the troublesome primitives can be accomodated better and how generational collection can be done in a simple and intuitive way. However, where Touati and Hama still wish to retain properties such as memory recovery on backtracking, we take a more radical approach: ease of garbage collection is more important than recovering memory on backtracking.

Bekkers, Ridoux and Ungaro [15] observe that it is possible to reclaim garbage collected data on backtracking if copying collection starts at the oldest choice point (bottom-to-top). Their method has several differences to ours.

- Their algorithm does not preserve the heap order, which means primitives such as `@</2` will work incorrectly. They do not indicate how this problem should be solved.
- Their algorithm (the version that incorporates early reset) copies data twice, while our algorithm visits data once and then copies the visited data. We think our approach leads to better locality of reference. However, we have not found any published measurements of the efficiency of the Bekkers-Ridoux-Ungaro algorithm.
- Variable shunting [83, 116] is used to avoid duplication of variables inside structures. This may introduce new variable chains, as shown in Paper D. We want to avoid this situation.

Their algorithm does preserve the segment structure of the heap (but not the ordering within a segment). Hence, they can reclaim all memory by backtracking. In contrast, our algorithm only supports partial reclamation of memory by backtracking. Our measurements indicate that this is sufficient: the copying algorithms we describe do not reclaim appreciably less

memory on backtracking than the standard mark-sweep algorithm on the measured benchmarks.

Subsequently, Demoen, Engels and Tarau [52] proposed and implemented an extension of the Bekkers-Ridoux-Ungaro bottom-up copying algorithm, combined with our mark-copy algorithm. They note that instant reclaiming of segments can be improved by moving data between segments. If some datum resides in segment i but is reachable only from a younger segment $i+n$, it can be migrated to segment $i+n$ and potentially be reclaimed earlier than would otherwise be possible. This relies on trimming choicepoints to delete dead fields. Furthermore, they show that top-down copying can lead to more major collections than bottom-up copying, since it can leave garbage in the old generation. We note that if the old generation is mostly filled with garbage, then a major collection will be quick, since it touches only live data. They find that the loss of instant reclaiming due to not retaining the heap ordering at a collection is "not prohibitive", which appears to confirm our results. Demoen et al do not directly compare their algorithm's efficiency with ours. However, their mark-copy algorithm appears to be approximately as efficient as ours. They do not present measurements for generational mark-copy.

Sahlin [115] has developed a method that makes the execution time of the Appleby et al. [8] algorithm proportional to the size of the live data. The main drawback of Sahlin's algorithm is that implementing the mark-sweep algorithm becomes more difficult. To our knowledge it has never been implemented. We also believe that Sahlin's algorithm is not as efficient as ours since it requires an extra pass over the live data, in addition to the passes of the Appleby algorithm. Since our algorithm is almost 70 % faster than the Appleby algorithm even when the heap is filled with live data, it is unlikely that Sahlin's algorithm will be more efficient than ours.

Papers E,F and G

And-parallelism A parallel logic programming system exploiting dependent and-parallelism as well as or-parallelism was designed by Conery and Kibler [38], though with impractical overheads.

De Groot [49] and Hermenegildo [67] restricted parallel execution to independent and-parallelism, by performing runtime tests to determine whether a given conjunction is to be run in parallel or sequentially. When goals do not share variables, they can be executed in parallel. Backtracking can then be managed locally, possibly followed by killing all sibling goals when one goal in a parallel conjunction failed. Systems that rely solely on independent and-parallelism have been investigated by Hermenegildo [67, 68, 69] and others. In general, the performance is quite good when independent and-parallelism can be exploited. The overhead for parallel execution is mainly

related to process management and runtime tests for independence. An optimizing compiler can reduce the overhead of independence tests [102, 28].

Subsequent work combined independent and-parallelism with or-parallel [63] and dependent and-parallel [60] execution to extend the scope for parallel execution. The execution engines for the combination proposals are far more complex than a sequential Prolog engine, which is unfortunate if the goal is high absolute speedups with respect to a sequential system.

Yang [162] defined syntactic conditions for programs for parallel execution and found that deterministic computations were well-suited for parallel execution, since conflicting bindings meant that part of the computation failed rather than backtracked. Warren [160] formulated the related *Andorra principle*, where deterministic goals are executed in parallel and nondeterministic goals are suspended. Determinism is detected at runtime. Yang's static method was thus replaced with a dynamic method of detecting and exploiting parallelism. This approach is also called *determinism-driven parallel execution*. An implementation of the Andorra principle, Andorra-I, is described in several papers [118, 119, 120, 163].

Shen [129, 127] developed a method that allows *general* dependent and-parallelism, by maintaining enough information to restart processes that have consumed bindings that are then invalid, and by restricting access to shared variables. The resulting engine executes at about 25% the speed of SICStus Prolog, a standard Prolog implementation, and provides almost transparent parallelization of Prolog programs.

A different approach to binding conflicts was proposed by Naish [103] with PNU-Prolog. By requiring programs to be *binding determinate*, i.e., not undo any bindings during parallel execution, the binding conflict problem was once again reduced to global failure. Binding determinism was a major influence on the Reform Prolog execution model.

Clark and McCabe introduced explicitly concurrent language constructs in IC-Prolog [32] and further developed by Clark and Gregory in their work on the Relational Language [33]. From this school of thought sprang three *concurrent logic languages*, Concurrent Prolog [126], Parlog [34] and Guarded Horn Clauses (GHC) [152]. A restriction of the latter was chosen as the base language of the Japanese Fifth Generation Project. They all had in common that nondeterminism was strictly curtailed by suspending until only one alternative was possible or committing arbitrarily to one alternative when several were available. A consequence was that binding conflicts caused global failure rather than backtracking. (Certain proposed metaprimitives allow programmers to encapsulate a computation to recover from failure.) The Andorra Kernel Language [64] was proposed to join concurrent logic programming with Prolog's don't-know nondeterminism by splitting nondeterministic states into several deterministic computations.

A different approach is to provide the programmer with powerful linguistic constructs to specify large, regular parallel computations. The data-parallel paradigm, where the user specifies a single thread of control that operates simultaneously on many data objects, is suitable for this purpose.

Barklund and Millroth [11] constructed Nova Prolog, a data-parallel logic language, which later was generalized into *bounded quantifications* [12]. A bounded quantification expresses some action to be taken over a finite set of elements, which often allows data-parallel execution [10]. Voronkov [157] independently laid theoretical foundations for bounded quantifications. Using bounded quantifications allowed a concise data parallel approach to logic programming. Finally, Sehr [123, 124] has independently proposed extraction of data-parallelism in the context of Prolog; see also Section 1.3.

Loop parallelization Extracting parallelism from loops is the most popular method to parallelize imperative programs. There are essentially two methods: vectorization and concurrentization [161]. Vectorization translates program statements into vector instructions for supercomputers, and we will not consider it further. Concurrentization entails adding synchronization primitives to the loop body to ensure correct execution. Synchronization requirements are derived from *dependence analysis*, which determines what loads and stores can interfere. Recently, array dataflow analysis [54] has been developed to deal more precisely with this problem.

Automatic loop-level parallelization of imperative programs has met some difficulties. For instance, Banerjee et al [9] note:

> "Dependence analysis in the presence of pointers has been found to be a particularly difficult problem. ...Much work has been done on this problem, though in general it remains unsolved."

Reform Prolog also exploits loop level parallelism, but parallelizes programs with arbitrary pointer structures – the clean semantics and execution model of the language makes the required analyses simpler than in the imperative case. Singh and Hennessy [131] note that parallelization technology is too limited. In particular, loop parallelism by itself is insufficient to extract enough parallelism from the examined programs. We believe this is in part due to the inability to parallelize loops with procedure calls. Reform Prolog allows procedure calls in parallel procedures and checks statically that they conform to the execution model.

Harrison [65] developed a system that parallelizes recursive calls in Scheme. In particular, the implementation of recursion-parallelism is similar to ours, though presented in a more general context. Harrison performs side-effect and dependence analysis to restructure and parallelize Scheme programs.

The main differences of our system are (i) that Prolog relies on 'side-effects' in the form of single-assignment to a much higher degree than Scheme, which necessitates a different treatment from that described by Harrison, and (ii) that Harrison considers only **doall** loops and recurrences, which do not require the explicit insertion of synchronization instructions. The Reform compiler handles general **doacross** loops. Harrison performs a thorough job of restructuring the computation, which could be a useful future extension to our system.

Larus and Hilfinger describe an advanced parallelizing compiler for Lisp, Curare [82], that performs alias analysis prior to restructuring [81]. By computing the program dependence graph of the program, they can extract parallelism from the program. The alias analysis then serves as a data dependence analysis. Reform Prolog also computes an alias analysis, but extracts parallelism less freely. Furthermore, Reform Prolog reasons about locality of data, which Curare apparently does not. Locality information turns out to be very useful when processes are medium or coarse grained. Interestingly, Curare uses a *destination-passing* technique to improve parallelism, reminiscent of logic variables. The destination-passing style means a function takes an extra argument where the result of the function is stored.

Sehr [123, 124] has proposed a parallel Prolog system, built on the same principles as Reform Prolog. His system exploits or-parallelism and (essentially) recursion-parallelism, but is restricted to and-parallelizing only 'inner loop' predicates. In contrast, our compiler considers the entire program and can insert synchronization and locking instructions in predicates called from the parallel loop, not only the parallel loop predicate itself. Sehr's system was, as far as we know, never implemented.

Static analysis The dataflow analyzer of Reform Prolog was built by extending a framework proposed by Debray [45]. The abstract domain used is an extension of that of Debray and Warren [46, 43]. While their domain tracked may-aliases and modes, Reform Prolog tracks modes and list types (including some difference lists), may- and must-aliases, linearity, locality and determinism.

Aquarius Prolog [154] used a simple mode analysis to improve sequential code. While Aquarius Prolog's domain does not fully track aliases, and so is somewhat weaker than Reform Prolog's domain, we have found that on a number of programs, the list types of Reform Prolog are the sole practical difference in precision. Taylor [142, 143] proposed a somewhat stronger domain, based on depth-k tracking of term structure, modes, constant types and recursive list types, but did not publish any precision results. Getzinger [58] found that a domain similar to Taylor's provided the best cost-benefit ratio for sequential compilation, out of a large collection of domains.

The treatment of aliases in Reform Prolog is similar to that of Chang's SDDA [28], in that aliases are treated as equivalence classes of possibly or certainly aliased variables. Subsequently, more powerful tracking of aliases has been proposed, e.g., by Muthukumar and Hermenegildo [102], Sundararajan [134], Jacobs and Langen [71] and the PROP domain [39, 153]. These proposals have been used for accurate groundness information, however, while Reform Prolog uses aliases to keep track of freeness.

The linearity domain of Reform Prolog keeps track of whether a term contains repeated occurrences of variables [73]. The approach taken by Reform Prolog is less sophisticated than other proposed domains [73, 135, 35, 72], but appears to work well in practice. This is probably because many Prolog programs do not exhibit complex sharing.

1.4 CONCLUSION

The contributions of this thesis span a wide range of topics, centered around compiled execution of Prolog.

- We have developed a continuation-passing style that fully compiles the control of Prolog.
- We have developed a simple and efficient method for constructing the control flow graph of Prolog programs.
- We have developed a simple, efficient and robust technique to detect uninitialized arguments to Prolog predicates.
- We have adapted copying and generational copying garbage collection techniques to Prolog and demonstrated that these techniques are more efficient than traditional collectors.
- We have developed an efficient execution model for dependent and-parallel and recursion-parallel execution of Prolog programs.
- We have demonstrated that a sophisticated compiler can substantially reduce suspension and locking overhead in a dependent and-parallel Prolog implementation.

A Continuation-Passing Style for Prolog

Thomas Lindgren

In *Proc. International Logic Programming Symposium 1994*, MIT Press, 1994.

We propose a translation of Prolog (including cut, if-then-else and similar constructs) into a first-order continuation-passing style, where (almost) all predicates have one clause and all clauses are binary. The resulting programs can be used as a flexible intermediate representation for compilation with a number of advantages as compared to Warren's abstract machine and other proposals. We discuss the advantages and disadvantages of this approach.

2.1 INTRODUCTION

The purpose of this paper is to define an intermediate representation for the compilation of Prolog and related languages. Previous research in the functional language community has shown that the continuation-passing style (CPS) may express the control of a computation much in the same way as control flow graphs do in imperative languages. This is promising for advanced compilation, and has been successfully exploited in a number of compilers [136, 79, 5].

In this paper, we propose a compilation of first-order Prolog programs into continuation-passing Horn clauses. The resulting predicates are first order and have a completely deterministic control: all control decisions have been moved to the source level. The final program can be viewed as a number of primitive operations, interconnected by jumps, continuation construction and continuation invocation.

We review related work in Section 2.2, describe the continuation passing transformation in Section 2.3, and several optimizations in Section 2.4. The

paper is concluded with a discussion of advantages and disadvantages of the proposed representation in Section 2.5.

2.2 RELATED WORK

There has been considerable interest in transformations of Prolog into Scheme and partial continuation passing styles and the relation of Prolog to intermediate representations, such as Warren's abstract machine [159]. We review earlier research in this section.

Warren's abstract machine

As an intermediate format, it is interesting to compare our intermediate format to Warren's abstract machine, WAM [159].

- Our notation extends the scope of compilation (the 'chunks' or 'basic blocks') by considering actions taken on failure.
- Indexing can be expressed as a source-to-source transformation and also extended to utilize other mutually exclusive tests to restrict the number of candidate clauses.
- We can trade code size against possible unnecessary data allocation by intelligent generation of continuation clauses. Our formulation admits both the approach used in WAM, that of simple binarization, as well as a spectrum of trade offs in between.
- Continuation representation is explicit, which allows trimming of choice points as well as environments. In the WAM, access is restricted to the most recent choice point and environment. Our notation represents these as terms, which means a compiler can change the parameter passing conventions to suit particular situations, alter the representation of choice points, and so on. This is further discussed in Sections 2.4-2.5 below.
- Internal registers are saved separately from argument registers, which implements shallow backtracking by default rather than as a special optimization. We can also admit shallow backtracking in several levels by nested saving of internal registers.

On a more fundamental level, the WAM hides much of the control flow of the computation from the compiler writer. This simplifies compilation but excludes some optimizations. Our representation directly admits a translation into control flow graphs, since control flow is explicit.

Translations in Prolog

Our technique can be viewed as an extension of binarization as described by Tarau [139] and Demoen [51, 50]. Binarization translates clauses

$$H \leftarrow$$
$$H \leftarrow B_1, B_2, \ldots, B_n$$

into their (success-) continuation passing forms:

$$H(S) \leftarrow \mathit{call}(S)$$
$$H(S) \leftarrow B_1(B_2(\ldots(B_n(S))\ldots))$$

Thus, the control rule of SLD-resolution is made explicit. In this paper, we only consider 'simple' binarization as above, though we note that more advanced forms of binarization exist. The above form seems to be the one used in practice.

Binarization ignores the failure continuation that relates the different clauses of a predicate. The effects of this is that cuts are handled ad hoc, that control constructs other than conjunction must be handled by rewriting the program so that only conjunctions remain, and an implicit control stack remains (the stack of choicepoints). Our notation avoids these problems and also allows us to directly describe indexing as a source-to-source transformation, which binarization instead delegates to an underlying abstract machine.

Nilsson [106] shows how to derive the WAM choicepoint instructions by partial evaluation of a meta interpreter. Our work is distinct from the control aspects of the WAM and adds translations of other control constructs such as cut.

Brisset and Ridoux [24] propose a CPS for λProlog. Since this language has λ-abstractions, their translation is similar to Scheme or Lisp translations discussed below, but does not explicitly manage substitutions. We have found that explicit operations that save and restore state can be useful, since they then can be moved by program transformations. An example is given below, where a predicate is recognized as deterministic and a pair of state save/restore operations can be removed altogether.

Since they generate higher-order terms in their translation, a subsequent pass of closure conversion [5] converts the term to a first-order representation. Our algorithm directly generates first-order terms as continuations.

Brisset and Ridoux also show how to translate exceptions by continuation capturing primitives. Such operations are beyond the scope of this paper.

Optimization of Prolog using control flow graphs

Debray [44] proposes several optimizations based on Prolog predicates translated into control flow graphs with success and failure edges. These edges do not correspond to the actual control of the program, for instance if B_1 continues with B_2 on success and B_2 on failure jumps to B_3, a success edge is added from B_1 to B_3 even though B_1 will never directly jump to B_3. This framework is then used to formulate several optimizations, such as moving tag tests and dereferencings out of loops. We believe that, mutatis mutandis, these optimizations could be carried out on our compiled programs as well.

Translations into Scheme and Lisp

Several translations from Prolog into Lisp have been proposed. We take the work of Kahn and Carlsson as representative in this approach. Kahn and Carlsson [75, 25] propose the use of *upward failure* or *downward success* continuations. The former relies on using lazy streams to produce solutions, while the latter performs a procedure return on failure. We note that both methods to some extent bury the symmetry of success and failure continuations, and that neither method eliminates all control stacks. Kahn and Carlsson [76] then employ partial evaluation to produce quite good Lisp code for naive reverse with modes.

Research at Indiana University showed that Horn clauses could be translated into Scheme by fairly straightforward means [55, 77] and that some care was needed to extend Prolog with continuations [66]. In particular, a generalized trail was required to suspend and resume a proof tree branch. Such operations are beyond the scope of this paper; all our failure continuations obey a stack-like discipline.

2.3 TRANSLATION TO CONTINUATION PASSING STYLE

We assume predicates have been rewritten into a single clause

$$p(X_1, \ldots, X_n) \leftarrow B$$

This can be done by introducing disjunctions of clause bodies and explicit unifications. These single-clause predicates are then translated into a *binary continuation style*, or BCS for short.

Transformation algorithm. Given a term $A = p(X_1, \ldots, X_n)$, we shall write $A(T_1, \ldots, T_m)$ for $p(X_1, \ldots, X_n, T_1, \ldots, T_m)$. Given this, the algorithm for the BCS transformation is shown in Figures 2.1-2.2 and translates one single-clause predicate into a set of continuation passing clauses.

$$\text{compile}(H \leftarrow B) = \{H(S,F) \leftarrow B'(S,F)\} \cup C \quad \text{where} \quad \langle B' \mid C\rangle = [\![B]\!]_{F,\text{fv}(H)\cup\{F\}}$$

$$[\![A,B]\!]_{C,FV} = \left\langle ab(\overline{X}) \;\middle|\; \begin{array}{l} \{ab(\overline{X},S,F) \leftarrow A'(c(\overline{Y},S),F), \\ call(c(\overline{Y},S),F) \leftarrow B'(S,F)\} \\ \cup\, C_A \cup C_B \end{array} \right\rangle$$

where

$$\langle A' \mid C_A\rangle = [\![A]\!]_{C,FV\cup\text{fv}(B)}$$
$$\langle B' \mid C_B\rangle = [\![B]\!]_{C,FV\cup\text{fv}(A)}$$
$$\langle \overline{X}, \overline{Y}\rangle = \text{live_fv}(A,B,C,FV)$$

$$[\![A;B]\!]_{C,FV} = \left\langle ab(\overline{X}) \;\middle|\; \begin{array}{l} \{ab(\overline{X},S,F) \leftarrow \\ \quad save(N, A'(S, c'(\overline{Y},N,S,F))), \\ call(c'(\overline{Y},N,S,F)) \leftarrow \\ \quad restore(N, B'(S,F))\} \\ \cup\, C_A \cup C_B \end{array} \right\rangle$$

where

$$\langle A' \mid C_A\rangle = [\![A]\!]_{C,FV\cup\text{fv}(B)}$$
$$\langle B' \mid C_B\rangle = [\![B]\!]_{C,FV\cup\text{fv}(A)}$$
$$\langle \overline{X}, \overline{Y}\rangle = \text{live_fv}(A,B,C,FV)$$

$$[\![\,P\,]\!]_{C,FV} = \langle exec(P) \mid \emptyset\rangle\,,\ P \text{ primop}$$

$$[\![\,G\,]\!]_{C,FV} = \langle G \mid \emptyset\rangle\,,\ G \text{ user-defined procedure}$$

$$\text{live_fv}(A,B,C,FV) = \langle \overline{X}, \overline{Y}\rangle$$

where

$$\overline{X} = (\text{fv}(A) \cup \text{fv}(B) \cup \xi(A,B,C)) \cap FV$$
$$\overline{Y} = (\text{fv}(B) \cup \xi(c,B,C)) \cap (FV \cup \text{fv}(A))$$
$$\xi(A,B,C) = \{C\}, \text{if } (! \in A) \vee (! \in B)$$
$$= \emptyset, \text{otherwise}$$

We note that C is the cut continuation, FV the variables appearing outside the current construct, ab, c, c' are fresh predicate symbols and that $\text{fv}(\cdot)$ yields the set of free variables in the argument term. Symbol $\perp$ signifies that a cut appearing in the construct is a compile-time error.

Figure 2.1: BCS translation

$$
[\![A \rightarrow B_1; B_2]\!]_{C,FV} = \left\langle ab(\overline{X}) \,\middle|\, \begin{array}{l} \{ab(\overline{X},S,F) \leftarrow \\ \quad save(N, A'(\quad cut(F, c(\overline{Y},S)), \\ \qquad\qquad\qquad c'(\overline{Z},N,S,F)))), \\ call(c(\overline{Y},S),F) \leftarrow B_1'(S,F), \\ call(c'(\overline{Z},N,S,F) \leftarrow \\ \quad restore(N, B_2'(S,F))\} \\ \cup\, C_A \cup C_{B_1} \cup C_{B_2} \end{array} \right\rangle
$$

where

$$
\begin{array}{lcl}
\langle A' \mid C_A \rangle & = & [\![A]\!]_{C,FV \cup \mathrm{fv}(B_1) \cup \mathrm{fv}(B_2)} \\
\langle B_1' \mid C_{B_1} \rangle & = & [\![B_1]\!]_{C,FV \cup \mathrm{fv}(A)} \\
\langle B_2' \mid C_{B_2} \rangle & = & [\![B_2]\!]_{C,FV \cup \mathrm{fv}(A)} \\
\langle \overline{X_1}, \overline{Y} \rangle & = & \mathrm{live_fv}(A, B_1, C, FV) \\
\langle \overline{X_2}, \overline{Z} \rangle & = & \mathrm{live_fv}(A, B_2, C, FV) \\
\overline{X} & = & \overline{X_1} \cup \overline{X_2}
\end{array}
$$

$$
\begin{array}{lcl}
[\![A \rightarrow B]\!]_{C,FV} & = & [\![A \rightarrow B; \mathrm{fail}]\!]_{C,FV} \\
[\![\neg(A)]\!]_{C,FV} & = & [\![A \rightarrow \mathrm{fail}; \mathrm{true}]\!]_{\perp,FV} \\
[\![\,!\,]\!]_{C,FV} & = & \langle cut(C) \mid \emptyset \rangle
\end{array}
$$

Figure 2.2: BCS translation (continued)

The basic idea behind the BCS transformation is the following: a continuation is coded as a term, which is subsequently used to invoke the correct procedure by switching on the term functor. The translation generates new continuation terms and their corresponding clauses (call/1 and call/2 for failure and success continuations, respectively) during 'compilation'. These are not to be confused with the Prolog higher order builtin call/1.

There are also some primitive operations: exec/3 executes a Prolog builtin operation and invokes the success or failure continuation; primitives save/2 and restore/2 saves the current substitution and replaces the current substitution by a stored one, respectively, and then continue with their respective continuations.

The generated programs have the following properties.

- Primitive operations P are encapsulated in exec(P,S,F).
- Nondeterministic choices are coded by (a) saving the current state, (b) building a continuation that restores the saved state, and (c) passing this as a failure continuation.
- All user-defined, 'entry point' predicates are defined by a single, binary clause $p(X_1, \ldots, X_n, S, F) \leftarrow B$. They take two extra arguments: a success continuation S, and a failure continuation F. They can be

viewed as 'labels with arguments,' since head unifications are trivial and they are deterministic.

- The call/1 and call/2 clauses are mutually exclusive and have linear heads with a single nonvariable term $c(X_1, \ldots, X_n)$ as first argument. They can be viewed as implementing two jump tables, or as code pointers on the implementation level.

There is a small number of predefined call/$\{1,2\}$ clauses. We show the two most interesting ones.

```
call(cut(C,S),F) :- call(S,C).
call(exec(G,S),F) :- exec(G,S,F).
```

Others include clauses for true/0, fail/0 and so on.

An example. To give a flavour of the structure of generated code, we show a small example.

```
member(A,B) :-
   ( B = [A|_]
   ; B = [_|C], member(A,C)
   ).
```

After transformation into continuation passing form, it becomes:

```
member(A,B,S,F) :-
    save(X, exec(B=[A|_],S,
                 mem1(X,A,B,S,F))).
call(mem1(X,A,B,S,F)) :-
   restore(X,exec(B=[_|C],member(A,C,S),F)).
call(member(A,C,S),F) :-
   member(A,C,S,F).
```

Note that mem1/5 serves a dual role: if the exec/3 in member fails, mem1/5 is simply invoked (or 'jumped to'). On the other hand, if the exec/4 succeeds, mem1/5 is actually constructed as a failure continuation (data structure) which is passed to S.

Let us consider the code that will be generated for a WAM given the same predicate, compared to straightforward code based on the definition above. The code is shown in Figure 2.3. Note that the BCS notation optimizes shallow backtracking by default: a choice point is not created until the unification is finished. There are also numerous opportunities to reuse results

WAM		BCS	
member:		member:	
	try L1		save internal regs in X
	trust L2		unify(L1,L2)
L1:	*unify*	L1:	build fail cont(L2,*regs*,*X*)
	proceed		invoke succ cont
L2:	*unify*	L2:	restore internal regs from X
	execute member/2		unify(member/2,L3)
		L3:	invoke fail cont

Figure 2.3: Stylized code for WAM and BCS

on a low level: saving internal registers can be delayed until the choice point is built, as long as the first unification does not overwrite them (instead, these registers can be duplicated and the copies overwritten); moreover, the unifications at L1 and L2 can be connected: if we find at L1 that the second argument is a list, this check need not be repeated at L2; nor need dereferencings or other such operations be repeated. Exploiting these opportunities may entail a code size increase; however, a clever compiler could use them without undue difficulty if they are found to be profitable.

Continuation trimming. The basic translation trims the size of continuations by only including the variable names that are visible within as well as outside the continuation. Consider the following predicate:

```
p(X,W) :- q(X,Y), r(Y,Z), s(Z,W).
```

It becomes the following clauses after compilation:

```
p(X,W,S,F) :-
   qrs(X,W,S,F).
qrs(X,W,S,F) :-
   q(X,Y,rs(Y,W,S),F).
call(rs(Y,W,S),F) :-
   r(Y,Z,s(Z,W,S),F).
call(s(Z,W,S),F) :-
   s(Z,W,S,F).
```

Note that only the 'live' variables are passed in the continuations rs/3 and s/3 and procedure qrs/5. If the optimization was not used, each of the above would have $\{X, W, Y, Z\}$ as arguments, in addition to the continuation and

substitution arguments. In practice, we also need to ensure that the cut continuation is passed along when required.

This generalizes environment trimming to apply to any continuation. In particular, choice points are trimmed similarly when applicable. In the WAM, environments can be physically updated to remove useless variables; in our representation, a new structure is created per continuation. We conjecture that simple forms of compile-time garbage collection and escape analysis (to tell whether continuations can be stack allocated) may be used with runtime checks to achieve the same effect.

2.4 OPTIMIZATIONS

Code sharing. A BCS compiler can regulate code size by generating more or less specific continuations. In the WAM, a separate piece of code (corresponding to a single BCS clause) is generated for each goal in the clause body. This avoids writing unnecessary data to the heap in case of failure. Simple binarization instead translates the clause body into a single call with a large continuation term.

$$G_1(G_2(\ldots(G_n(S))\ldots))$$

This reduces code size, since terms can be written compactly, but at the expense of speculatively creating continuation terms.

The basic BCS translation follows the WAM approach, but can be adjusted to share code by two means. First, the compiler can generate simple continuation clauses and speculatively create data. Consider the second clause of the nrev/2 predicate:

```
nrev([X|Xs],Zs) :- nrev(Xs,Ys), append(Ys,[X],Zs).
```

We can generate the following BCS code for the clause:

```
nrev2(A,B,S,F) :-
  exec(A=[X|Xs],nrev(Xs,Ys,app(Ys,[X],B,S)),F).
call(app(A,B,C,S),F) :- append(A,B,C,S,F).
```

Now the definition of app/4 can be used for all calls to append/5, which is what simple binarization does; the term [X] is however created speculatively, since it is unnecessary work if nrev/4 fails. A spectrum of trade-offs is possible, by varying the amount of data that are created speculatively.

The second technique is to share larger sequences of goals (e.g., all occurrences of some goal $B_1; B_2; B_3$) by generating a single continuation clause. (Below, $\overline{X}$ is the set of live free variables computed as previously.)

$$\mathit{call}(\mathit{bbb}(\overline{X}, S), F) \leftarrow [\![B_1; B_2; B_3]\!]$$

The most obvious use of this method is for indexing, where the compiler may share code for a disjunction of clauses or tests for selecting a collection of clauses. This technique converts a decision tree into a decision graph, which is needed to make advanced indexing techniques practical. We note that one can equally well use the method to share code for common conjunctions of goals.

Grouping primitives. The basic BCS algorithm generates small definitions, in order to avoid code duplication (and for simplicity of exposition). To increase the scope of compilation, we would like to group primitive operations into chunks of operations, moving together primitives executed on success as well as failure.

This can be done by unfolding after generating BCS code, or by generating larger definitions immediately. The BCS algorithm is then modified to group together consecutive primitives performed on success (the same optimization is done by WAM compilers) and possibly to group primitives performed on failure.

Grouping primitives on failure has the disadvantage of code duplication (or executing unwieldy continuation terms). Assume that P_i are primitives and G_j are user goals. Consider the following clause body:

$$(P_1, P_2, G_1; P_3, P_4, G_2)$$

If P_1 or P_2 fails, we would like to directly reset the saved state and jump to P_3. Hence, we could generate the code below. (This code is not generated by the previous BCS algorithm.)

```
save(X,
exec(P_1,exec(P_2,G_1(S)),
     restore(X,
      exec(P_3,exec(P_4,G_2(S)),F))))
```

However, the failure continuation passed to G_1 will then be the unwieldy restore/2 term, which essentially interprets which primitives to execute. For efficiency, the compiler should rather generate a continuation clause and pass the appropriate failure continuation:

```
save(X,
exec(P_1,exec(P_2,cut(c(...,X,S,F),G_1(S))),
     restore(X,
      exec(P_3,exec(P_4,G_2(S)),F))))

call(c(...,X,S,F)) :-
```

```
restore(X,
 exec(P_3,exec(P_4,G_2(S)),F)).
```

But now the code for (P_3, P_4, G_2) has been duplicated. The problem is that the 'label' of the second disjunct is captured at the call to G_1 but also used as part of a larger scope of compilation.

A common case (e.g., in implementing indexing) is a disjunction of if-then-else operations, where the failure continuation is cut away, which avoids this problem. Consider the clause body $((P_1, P_2) \rightarrow G_1; P_3, P_4, G_2)$. The generated code can be freely unfolded since the cut operation removes the need to generate an extra call-clause.

```
save(X,
exec(P_1,exec(P_2,cut(F,G_1(S))),
     restore(X,
     exec(P_3,exec(P_4,G_2(S)),F))))
```

In conclusion, unfolding conjunctions of primitives is feasible, while unfolding disjunctions runs the risk of code explosion; if-then-else operations remove the need for code duplication in disjunctions.

Indexing. Indexing in the WAM is done in three levels: on the type of the first argument, on the actual functor or atom of the argument (if applicable) and finally, by generating specialized disjunctions of the applicable clauses. We note that these optimizations can be expressed at the Prolog source level instead, by using the appropriate primitive operations (type tests for first level indexing and functor/3 for testing the actual values) and if-then-else to encode deterministic choice.

In the WAM, all specialized try/retry/trust sequences share code for the clauses (the original WAM scheme even shares some try/retry/trust sequences). One way of sharing disjunctions in BCS is shown above as **code sharing**.

The advantages of doing indexing this way is flexibility: indexing is easily done on any argument, and may intersperse other tests than those done in the WAM. An enterprising implementation could generate full decision graphs and stack multiple save/restore operations if needed. Join points in the decision graph are then encoded by call-clauses; control flow analysis [130] will reveal the definitions calling these clauses and may enable a compiler to take advantage of the contexts of the callers. (Since we are discussing first-order Prolog without freeze/2 etc, control flow analysis is straightforward; space restrictions do not permit us to describe the algorithm used here.)

Motion of save/restore operations. Explicitly saving and restoring the state has the advantage of allowing us to move these operations around. In particular, there is a quite simple optimization that allows us to delay or avoid saving the state when performing tests.

Consider the clause body $(X < N, B_1; X \geq N, B_2)$. It can be translated into the following code:

```
save(St,
 exec(X < N, B_1(S),
      restore(St,
       exec(X >= N, B_2(S),F))))
```

Since $X < N$ is a pure operation, we can move it past the save/2 operation into the failure branch. (It is unnecessary to also move it into the success branch.)

This in turn yields a code sequence *save(St,restore(St,B))*, which (since $St \notin B$) can be simplified into B.

```
exec(X < N, B_1(S),
     exec(X >= N, B_2(S),F))
```

A compiler could also infer that the test $X \geq N$ will always fail when entered from B_1 and always succeed when $X < N$ fails. Hence, we can generate the desired code.

```
exec(X < N, cut(F,B_1(S)),
            B_2(S,F))
```

The Aquarius compiler [154] performs this optimization by collecting test sets and checking whether the sets are mutually exclusive[1]. In our example, we split this into two parts: motion of save/restore operations, and finding mutually exclusive tests.

Continuation representation. The BCS representation gives us freedom to vary the representation of a continuation and access to the continuation. The WAM [159] and BinWAM [141] both use a 'flat' representation of an environment: in the WAM, all variables are accessed by offsets from a pointer into the stack, while the BinWAM simply creates continuation terms that correspond to flat environments.

[1]Correction (1996): Test sets are actually mutually exclusive by definition, though the Aquarius compiler still uses them to find deterministic computations.

Since environments are explicit, a BCS compiler can manipulate the representation of a continuation freely. For instance, free variables can be retrieved from linked environments instead of flat ones; several continuations may share a single environment (in particular, some success and failure continuations might share environments, e.g., in the style of Appel, where closures can share share bindings [5]). Other possibilities are discussed below.

2.5 CONCLUSION

We have shown how to compile Prolog procedures into BCS, a continuation-based intermediate form (which happens to be directly executable).

The advantages of BCS include extending the scope of optimization by compiling primitive operations grouped across failure branches and choice point saves; generalized trimming of both environments and choice points; automatic shallow backtracking optimization by decoupling internal registers from argument registers and by allowing internal registers to be assigned physical registers more flexibly; automatic removal of any unneeded cut register; and possible extensions to indexing and reasoning about primitives.

On a lower level, BCS allows us to view the program as a control flow graph, in the conventional imperative style. Thus, conventional compiler technology and Prolog compilation may be unified. This is similar to continuation-passing functional programs, though we use a first-order framework.

There are some drawbacks as well. First, some of the optimizations are less straightforward than in WAM. For instance, overwriting an environment (trimming it) requires some fairly sophisticated analysis. Our representation directly admits only declaratively installing a new, smaller environment instead.

Implemented straightforwardly, a BCS program allocates all data on the heap. Research in functional languages has shown that this is not necessarily worse than stack allocation given generational copying garbage collection and fast handling of write-misses in the cache [53, 7]. Bevemyr and Lindgren have previously shown how to adapt generational copying garbage collection to a standard WAM [21]; we expect to reuse that method. Shao and Appel have shown how to optimize continuation representations for a heap based implementation of SML [125, 6]. If their results can be applied to our representation, we can pass arguments in memory when registers are scarce, or pass continuations in registers when registers are plentiful. Tarau has shown that allocating WAM environments on the heap simplifies the execution machinery while being competitive in execution speed to a standard WAM implementation [140, 141]. Finally, some continuation-passing com-

pilers reintroduce stack allocation by escape analysis [136, 79]. In summary, we do not find default heap-only allocation a fatal flaw.

Future work. There are several issues to be explored. First, Prolog includes a great number of features, such as higher order predicates, dynamic predicates, input/output, coroutining and so on. It may be useful to express these in the same explicit way as done with control here. For instance, findall/3 may be coded with an explicit database parameter.

Second, explicit failure continuations can express some otherwise unwieldy transformations. Partial evaluators for Prolog require quite complex management of cuts; explicit cut continuations may reduce the need for such treatment. Likewise, intelligent backtracking can be expressed at a high level by passing multiple failure continuations. A full treatment of intelligent backtracking is likely to involve global analysis, some carefulness in passing failure continuations correctly and possibly unravelling primitive operations (such as unifications) to directly connect the cause of failure to a jump to the correct backtrack point.

Third, we have described an intermediate representation, rather than compilation into efficient programs. The purpose of such a representation is further compilation. As noted by earlier research, continuation-passing programs may be interpreted as control flow graphs (indeed, this was one of the motivations for this work). We will investigate the usefulness of conventional compiler technology to Prolog in the framework of continuation-passing programs.

Acknowledgements. My thanks to Håkan Millroth for good advice, to numerous members of the Computing Science Department for reading various versions of this paper and to Paul Tarau, Ulrich Neumerkel, Jacques Noyé and D.H.D. Warren for stimulating discussions. I would also like to thank the anonymous referees for valuable comments.

Control flow analysis of Prolog

Thomas Lindgren

In *Proc. International Logic Programming Conference 1995*, MIT Press, 1995.

The paper describes how to translate a Prolog program into a control flow graph (CFG). The program is annotated with control information and a system of simple set constraints is formed and solved. Using this information, a CFG can be constructed straightforwardly, which is described.

In order to represent control information compactly, the paper proposes a method to structure the solution of the analysis to share information. Compressing the control information reduces the size of the control information at least by half for 16 out of 26 benchmarks, and for some programs by an order of magnitude or more.

Finally, the control flow of a set of benchmarks is measured. Forward control turns out to be relatively simple, with typically 5 or less jump targets per predicate. In several smaller benchmark programs, all or close to all success returns could be converted into direct jumps.

Failure control is more complex, though there are typically 10 or less jump targets per predicate for the benchmark set.

3.1 INTRODUCTION

Prolog programs spend much of their time in control management [144, 148, 56]. Since procedure calls are very common, much effort is spent in saving and restoring parameters from the stack; furthermore, a procedure call typically requires some shuffling of parameters, as well as building new arguments. When backtracking may occur, the machine state is saved in a record; when a computation fails, the saved state is restored from such a record.

Compilers for Prolog and related languages have so far mostly taken a predicate-level view of compilation, even if global analysis has been used to improve the compilation of each predicate, and concentrated their efforts on reducing the cost of primitive operations. (Exceptions include call forwarding [23].) Perhaps unsurprisingly, the greatest improvements have been in smaller programs, where primitive operations are relatively common and the compiler has some scope to improve sequences of primitives. However, procedure calls have mainly been handled as in the WAM [159] (an exception is the stack-based WAM of Zhou [164]), using a fixed procedure calling convention and a pessimistic caller-saves assumption on what variables must be saved at a procedure call.

In this paper, we show how to take a **whole-program view** of compilation by converting an entire Prolog program into a control flow graph. This transformation reconciles the compilation of Prolog and logic programs with the compilation of imperative languages, since a compiler then can apply techniques for compiling Prolog as well as imperative languages to the program.

We contribute the following:

- Efficient control flow analysis by solving simple set constraints.
- Efficient construction of the control flow graph by factoring of control information.
- A study of the control flow properties of a substantial set of benchmark programs.

The paper is structured as follows. Section 3.2 discusses some preliminary program transformations. In Section 3.3, we introduce an explicit notation for control. Section 3.4 describes control flow analysis, which is instrumental in constructing the control flow graph. Section 3.5 shows how to construct the control flow graph. Section 3.6 evaluates the proposed analysis and measures the control flow information over a range of benchmarks. Section 3.7 discusses related work. Section 3.8 summarizes the main points of the paper and discusses future work.

3.2 PRELIMINARIES

We assume that all predicates are in *single-clausal form*, i.e., each predicate consist of a single clause where the head is a linear sequence of variables, and the clause body consists of disjunction, conjunction, if-then-else, cut and calls to primitive operations and user-defined predicates [86].

We furthermore assume that predicates have been converted into first-order programs (call/1 is assumed not to be present, or transformed away) and

do not use dynamic scheduling, calls to dynamic code or similar operations. Clause indexing is expressed by explicit type-tests and if-then-else operations, rather than by 'black box' instructions in the abstract machine.

Finally, we shall assume that the predicate `main/0` defines the starting point of the program, and `halt/1` terminates the program indicating success or failure. The actual compiler instead works with modules; this is straightforward but ignored for expository purposes.

3.3 MAKING CONTROL EXPLICIT

In the WAM, choicepoints encapsulate both the saved state of the computation and the argument registers to be restored. We separate this into two operations: save state (*save/1*), and save registers (*fail_cont/1*). Likewise, backtracking restores state (*restore/1*) and visible registers (*label/3*) in two explicit operations. Untrailing is done when the state is restored. The reason for this choice is that we can delay saving the registers or possibly avoid that operation entirely.

We first make state saving and cut operations explicit, by rewriting the program according to the table below. We also assume that cuts are handled the standard way: *save_cp(X)* saves a pointer to the current choicepoint and *cut(X)* sets the current choicepoint to X.

We assume that A rewrites into A', B into B' and so on.

Original	New
(A, B)	(A', B')
$(A; B)$	$save(X), (A; restore(X), B)$
$(A \rightarrow B_1; B_2)$	$save_cp(C), save(X),$
	$(A' \rightarrow cut(C), B_1'; restore(X), B_2')$

Then, we introduce continuation frames. At each procedure call, we compute the live and defined variables (i.e., the set of variables that occur in the predicate both prior to and after the call) as in Ref [86]. The first occurrences of variables local to the clause body are explicitly marked in order to generate reentrant code later on.

A continuation frame, or simply frame, $L[Xs]$ consists of a label L and a set of variables $Xs = \{X_1, \ldots, X_n\}$ to be restored when the continuation is invoked. It corresponds to a closure in a higher-order functional language, or a stack frame in an imperative language.

The operation *fail_cont(L[Xs])* creates a new failure continuation, holding *Xs* and linked with the previous success and failure continuations, while

succ_cont(L[Xs]) creates a new success continuation, linked with the previous success continuation. The operation *label(t,L,Xs)* unlinks the success ($t = s$) or failure ($t = f$) continuation, reloads the variables from the frame and continues execution.

1. Each non-last call to a user-goal G with live and defined variables Xs is rewritten into:

   $$succ_cont(L[Xs]), G, label(s,L,Xs)$$

2. A last call G in predicate p/n is rewritten into:

   $$succ_to(p/n), G$$

   to indicate that G will return to the caller of p/n.

3. Each disjunction $(A; B)$ where B has live and defined variables Xs is rewritten into:

   $$(fail_cont(L[Xs]), A'; label(f,L,Xs), B')$$

Explicit state save/restore and frame creation operations have the benefit that the compiler can remove them when redundant as well perform limited code motion on frame creations [87].

Consider the following annotated predicate. Some non-essential annotations have been removed.

```
p(X) :- save_cp(C), save(A),
        ( fail_cont(L1[A,X]),
          ( save(B), ( fail_cont(L2[B,X]), var(X) ;
                       label(L2),
                       restore(B), atomic(X) )) ->
          cut(C), p_simple(X)
        ; label(L1), restore(A),
          p_compound(X)
        ).
```

The compiler moves the L1 and L2 frames to the right, over non-procedure calls. Finally, the frames reach the cut(C) call and can be removed. (Note that there is some subtlety in telling when a frame can be removed and when it must be retained.) This produces the following code.

```
p(X) :- save_cp(C), save(A),
        (( save(B),
          ( var(X) ; restore(B), atomic(X) )) ->
          cut(C), p_simple(X)
        ; restore(A),
          p_compound(X)
        ).
```

Since var/1 and atomic/1 do not modify the heap, the save/restore sequence is unnecessary and the program is rewritten into the following.

```
p(X) :- save_cp(C),
        ( ( var(X) ; atomic(X) )) ->
          cut(C), p_simple(X)
        ; p_compound(X)
        ).
```

We note that this can be compiled into the desired tests and jumps efficiently. We could further eliminate *save_cp(C)* and *cut(C)*, though the gain is much smaller (two assignments, in essence).

3.4 CONTROL FLOW ANALYSIS

In order to find the global control flow of Prolog, we must know whereto a call will jump when it succeeds or fails. We solve this problem by computing

the set of labels to which a given predicate *may* jump on success or failure as a safe approximation to the exact solution.

The control flow analysis (CFA) is done in two steps.

1. Use the frames in the annotated program to derive local control flow constraints.
2. Solve the constraints to arrive at global control flow.

Local control flow

For every predicate p/n, we introduce three variables succ(p/n), fail(p/n) and redo(p/n). For a goal $G = p(T_1, \ldots, T_n)$, we sometimes write succ(G) to indicate succ(p/n), fail(G) to indicate fail(p/n) and so on. The variables denote the control flow when p/n succeeds, fails or is entered by backtracking, respectively, and range over sets of labels. The constraints have the form $x \supseteq S$ where x is a variable and S is a set $\{l\}$ or a variable. The meaning of the constraint is that S is a subset of x.

For lack of space, we show only some of the relevant rules in Figure 3.1. The remaining rules are similar or straightforward.

The constraints must approximate Prolog's left-to-right and top-to-bottom control rules. We motivate their definitions below.

Success constraints for a predicate $G = p(X_1, \ldots, X_n)$ are formed in two ways:

- G occurs as a last call in some predicate $q(Y_1, \ldots, Y_m)$. All success labels of q/m are also success labels of p/n due to tail-recursion optimization, so we form $succ(G) \supseteq succ(q/m)$.
- G occurs in a non-last call position. Hence, G is followed by a label l (due to the earlier annotation pass) and the constraint $succ(G) \supseteq \{l\}$ is formed.

We say that a label L from fail_cont(L[...]) or (the redo variable of) call H *reaches* a call G if there is a left-to-right path (according to Prolog's success control) from L or H, respectively, to G without passing a fail_cont/1 or predicate call.

Failure constraints are formed by considering the labels or calls that reach each call $G = p(T_1, \ldots, T_n)$ as targets for backtracking if G fails.

- If label L reaches G, the constraint *fail(G)* $\supseteq \{L\}$ is formed; the annotation phase ensures this is correct.

Success constraints are collected by inspecting the body B of every annotated clause.

- For each $(\mathit{succ_cont(L[_])}, G)$ in B, generate

$$\mathit{succ}(G) \supseteq \{L\}$$

- For each $(\mathit{succ_to}(q/m), G)$ in B, generate

$$\mathit{succ}(G) \supseteq \mathit{succ}(q/m)$$

Failure constraints are generated by traversing every annotated clause. $\mathsf{f}\,[\cdot]$ uses the set of variables and labels f that reaches each construct, and returns a set of constraints and a new set of variables and labels. The parameter c stores cut information.

Construct	Failure constraints
$\mathsf{cfa}\,[p(x_1, \ldots, x_n) \leftarrow B]$	$C \cup \{\mathit{redo}(p/n) \supseteq c \mid c \in f\}$ **where** $\langle C, f\rangle = \mathsf{f}\,[B](\{\mathit{fail(p/n)}\}, \{\mathit{fail(p/n)}\})$
$\mathsf{f}\,[p(t_1, \ldots, t_n)](c, f)$	$\langle C, \{\mathit{redo}(p/n)\}\rangle$ **where** p/n is a user-defined predicate $C = \{\mathit{fail(p/n)} \supseteq d \mid d \in f\}$
$\mathsf{f}\,[\mathit{fail_cont}(_, L[_])](c, f)$	$\langle \emptyset, \{\{L\}\}\rangle$
$\mathsf{f}\,[(A, B)](c, f)$	$\langle C_A \cup C_B, f_B\rangle$ **where** $\langle C_A, f_A\rangle = \mathsf{f}\,[A](c, f)$ $\langle C_B, f_B\rangle = \mathsf{f}\,[B](c, f_A)$
$\mathsf{f}\,[(A; B)](c, f)$	$\langle C_A \cup C_B, f_A \cup f_B\rangle$ **where** $\langle C_A, f_A\rangle = \mathsf{f}\,[A](c, f)$ $\langle C_B, f_B\rangle = \mathsf{f}\,[B](c, f)$
$\mathsf{f}\,[!](c, f)$	$\langle \emptyset, c\rangle$

Figure 3.1: Deriving local control flow constraints.

- If a call H reaches G, we form *fail(G)* $\supseteq$ *redo*(H), since if G fails, it may backtrack into H.
- If predicate q/m is entered by backtracking, it may backtrack to the labels or calls that reach the end of q/m. Hence, *redo*$(q/m) \supseteq C$, for all such C.
- Finally, a cut in predicate q/m means the reaching failure constraint variables are set to those of the beginning of the clause, simply $\{$*fail(q/m)*$\}$. This is consistent with the meaning of the cut operation.

Global control flow

The control information of interest for a given variable x is the least set of labels implied by the constraint set derived above. Our next problem is to compute this efficiently for all variables.

The set constraints can be solved efficiently by transforming them into a directed graph and computing the strongly connected components of the graph, e.g., using Tarjan's linear algorithm. Every constraint

$$c \supseteq c'$$

is translated into an arc

$$c \rightarrow c'$$

Assuming that l denotes a graph node corresponding to a label L in the program, the labels to which p/n can jump is the set $\{l \mid c(p/n) \rightarrow^* l\}$, where $c \in \{\text{succ}, \text{fail}\}$.

The labels reachable from every constraint node can be read efficiently from the reduced graph by caching the reachable labels per node in the reduced graph as they are computed. For each predicate p/n, the nodes of interest are succ(p/n) and fail(p/n).

Note that control flow analysis reveals unobvious entry points into the program, namely the alternatives pointed out by choicepoints returned to the toploop. Similarly, environments may escape when out-of-module calls are added.

Factored control information

The set of labels per node can be quite large and we thus run the danger of code explosion as we attempt to represent this information. Hence, it might seem to be unpractical to construct the control flow graph.

The structure of the reduced graph can be used to substantially factorize the control information, by making nodes share information.

First, it is clear that all nodes in a strongly connected component (SCC) reach the same set of labels. Hence, they can share the control information as well. Measurements (not shown in this paper) indicate a typical SCC is small for the programs we have encountered, so this measure does not save us much.

A second, quite effective trick is to use the *structure* of the reduced graph to further share information. We classify the SCC:s into the following categories:

- **Trivial.** SCC:s that only contain a single label, $\{l\}$, (and so point at no other SCC) are considered trivial. They constitute the leaves of the graph.
- **Jump.** Some SCC:s point only at a single trivial SCC. They can be implemented as jumps to the corresponding label.
- **Collapsible.** Many SCC:s s simply point out another SCC s'. Such 'towers' of SCC:s can be collapsed, so that s, s' and possibly other nodes share their (identical) control information.
- **Switch-default.** Other SCC:s point at a collection of trivial SCC:s, $\{\{l_1\}, \{l_2\}, \ldots, \{l_n\}\}$ and one non-trivial SCC s'. These can be represented by a node pointing out the corresponding labels $l_1, l_2, \ldots, l_n$, and 'falling through' onto the control information of s'.
- **Others.** Finally, some SCC:s actually point out two or more non-trivial SCC:s.

We measure the effectiveness of this classification in Section 3.6.

3.5 CONSTRUCTING THE CFG

A control flow graph (CFG) is a quadruple $\langle s, e, V, E\rangle$, where (V, E) is a directed graph, $s \in V$ is the unique start node and $e \in V$ the unique end node, such that there is a path from s to every node in V and there is a path from any node in V to e.

In constructing the CFG, we incrementally add nodes and arcs to the graph. We denote this by $\{n : c \rightarrow n' : c'\}$, where n and n' are added to V with contents c and c' respectively, and (n, n') is added to E. We take $\otimes$ to be a placeholder node, which will subsequently be filled in; this is used to anchor mutually recursive definitions. We shall assume that several primitive operations, such as assignment (:=), saving and restoring state (*save/1,restore/1*), and creating and consuming continuation frames (*succ_cont/1,fail_cont/1,reload/3*), are available. We trust their intended

meanings are clear. Furthermore, primitive operations are assumed to be accessible by procedure calls or the equivalent. The node *join* denotes a join point in the CFG.

The construction uses several mappings. *Global*(L) denotes the node to jump to when L is invoked by a continuation. *Local*(L) denotes the node to jump to when L is the target of a direct jump (e.g., local failure). *PredInfo*$(p/n) = \langle e, b, s, f, ps \rangle$ yields the entry point e of the predicate, the beginning of the body b, the success and failure exits s and f and the formal parameters ps. We begin with an empty graph.

1. Generate code for every label *label(t,L[*Xs*])*.
   - Add $\{n : \mathit{reload}(t, \mathit{Xs}) \rightarrow n' : \otimes\}$ to the CFG.
   - Set *Global*$(L) = n$ and *Local*$(L) = n'$.
2. For every predicate p/n, generate code for succ(p/n) and fail(p/n). The generated code straightforwardly follows the factored control representation: jump SCC:s are represented by jumps; switch SCC:s become switch-nodes, and so on. Use Global(L) to find the addresses of labels.
3. For every predicate p/n with head $p(X_1, \ldots, X_n)$ and s and f as code for success and failure, generate entry code.
   - Add $\{n : \mathit{join} \rightarrow n' : \otimes\}$ to the CFG.
   - Set *PredInfo*$(p/n) = \langle n, n', s, f, [X_1, \ldots, X_n] \rangle$
4. For every predicate p/n, generate code for the body. This process is a recursive walk over the syntax tree and is shown in Figure 3.2. Use *Local*(L) and *PredInfo*(p/n) to find nodes for labels and predicate calls.
5. Add *start* and *end* nodes using `main/0` and `halt/1`

3.6 EVALUATION

We used the above algorithms to compute the control flow graphs of a set of Prolog programs. As a benchmark suite, we chose the Berkeley benchmarks. Measurement data are shown in Figure 3.3.

The indexing algorithm has an impact on the results. In our experiments, we used a variation on first-argument indexing where we generated if-then-else code if a clause performed a shallow cut and switching code otherwise. The algorithm also duplicated simple clause bodies (consisting of primitives that did not create a choicepoint, possibly followed by a single procedure

Construct	CFG
$\mathsf{cfg}\ [p(X_1,\ldots,X_n) \leftarrow B]$	$C \cup \{b \leftarrow n\}$ **where** $\langle _, b, s, f, _ \rangle = \mathit{PredInfo}(p/n)$ $\langle C, n\rangle = \mathsf{g}\ [B](s,f)$
$\mathsf{g}\ [(A,B)](s,f)$	$\langle C_A \cup C_B, n_A\rangle$ **where** $\langle C_A, n_A\rangle = \mathsf{g}\ [A](n_B, f)$ $\langle C_B, n_B\rangle = \mathsf{g}\ [B](s,f)$
$\mathsf{g}\ [(A;B)](s,f)$	$\langle C_A \cup C_B, n_A\rangle$ **where** $\langle C_A, n_A\rangle = \mathsf{g}\ [A](s, n_B)$ $\langle C_B, n_B\rangle = \mathsf{g}\ [B](s,f)$
$\mathsf{g}\ [(A \rightarrow B;C)](s,f)$	$\langle C_A \cup C_B \cup C_C, n_A\rangle$ **where** $\langle C_A, n_A\rangle = \mathsf{g}\ [A](n_B, n_C)$ $\langle C_B, n_B\rangle = \mathsf{g}\ [B](s,f)$ $\langle C_C, n_C\rangle = \mathsf{g}\ [C](s,f)$
$\mathsf{g}\ [\mathit{cut(C)}](s,f)$	$\langle\{n : \mathit{fail} := C \rightarrow s\}, n\rangle$
$\mathsf{g}\ [\mathit{label(t,L[}\mathrm{Xs}\mathit{]})](s,f)$	$\langle\{n \leftarrow s\}, n\rangle$ **where** $n = \mathit{Local}(L)$
$\mathsf{g}\ [P](s,f)$	$\langle\{n : P \rightarrow s, n \rightarrow f\}, n\rangle$ if primitive P
$\mathsf{g}\ [p(T_1,\ldots,T_m)](s,f)$	$\langle\{n : [X_1 := T_1, \ldots, X_m := T_m] \rightarrow e\}, n\rangle$ **where** $\mathit{PredInfo}(p/m) = \langle e, _, _, _, [X_1,\ldots,X_m]\rangle$

Figure 3.2: Building the control flow graph. Rules for creating continuations, saving cut pointers and suchlike operations are similar to the rule for cut. Each call to $\mathsf{g}[\cdot]$ returns a sub-CFG and the entry node to this sub-CFG. The notation $e \leftarrow n$ means placeholder e is updated to n.

	succ arcs			fail arcs			factoring	
program	max	avg	med	max	avg	med	simple	factor
boyer	5	1.44	1	64	9.32	2	269	143
browse	2	1.12	1	6	3.31	2	71	31
chat	36	3.36	1	503	307	502	67894	1488
crypt	8	2.89	2	14	12.6	14	139	41
div10	5	2.33	1	1	1.0	1	10	10
fastmu	1	1.0	1	10	5.89	10	62	21
flatten	3	1.47	1	15	7.25	9	314	171
log10	5	2.33	1	1	1.0	1	10	10
metaqs	4	2.27	3	17	8.27	2	116	44
mu	2	1.7	2	8	4.7	7	64	40
nand	21	2.81	1	82	22.9	2	1943	922
nrev	2	1.6	2	3	1.8	2	17	14
ops8	5	2.33	1	1	1.0	1	10	10
poly10	9	3.43	3	5	3.64	5	99	50
prover	6	2.62	3	18	9.38	8	192	78
qsort	2	1.4	1	3	2.2	3	18	10
queens8	1	0.89	1	2	1.22	1	19	22
query	2	1.17	1	49	25	25	157	59
reducer	13	2.89	2	31	8.73	4	732	368
sdda	6	1.88	1	22	9.71	2	475	240
serial	2	1.33	1	7	4.17	6	66	34
simple	11	2.84	2	40	14.2	11	1504	678
tak	4	1.67	1	1	0.67	1	7	10
times10	5	2.33	1	1	1.0	1	10	10
unify	12	2.98	2	23	10.8	4	636	167
zebra	10	2.83	1	5	2.17	1.5	30	30

Figure 3.3: Measurement data: success control, failure control and the effect of factoring control

call) and broke out more complex clause bodies as new predicates when needed (e.g., due to being entered from several switching cases).

For the Berkeley benchmarks, the choice of indexing algorithm had a considerable impact on control flow information; several programs defeat first-argument indexing by being (shallowly) nondeterministic. An older version of the indexer, which performed first-argument indexing only and completely avoided code duplication by making clauses into new predicates, required considerably more control flow information than the present system.

The example programs also included numerous calls to out-of-module predicates; module exits had no outgoing arcs which accounts for, e.g., the less-than-one average in **queens_8**.

We measured the number of arcs (or *outdegree*) of success and failure exits per predicate. The outdegree for success control was mostly small, which is unsurprising since most predicates are called from a small number of places. In several smaller programs, all or almost all success returns from predicates could be determined to jump to a single program point (e.g., **queens_8**, **fast_mu** and **serialise**).

Failure control was more complex, though 20 out of 26 programs had an average outdegree of 10 or less per predicate. Only 3 out of 26 programs had an average failure outdegree of 15 or more. Five of the smaller benchmarks were found to be entirely deterministic, e.g. **div10** and its brethren.

We measured the effectiveness of control flow factoring. We computed this as follows: the **simple** representation does not share information at all, while the **factored** representation uses the scheme described previously to share information. In both cases, we measured the sum of outdegrees.

Factoring turns out to be quite successful. Eleven programs reduced the amount of control information substantially, by a factor two or more compared to the unfactored program. Another seven programs still saved 50% or more. Two programs actually increased the number of arcs; this is due to the implementation not collapsing if-then nodes where both cases point to the same node under special circumstances. This defect can be remedied.

Finally, we note that changing the indexing algorithm turned out to reduce control flow information substantially, by (a) improving indexing and (b) reducing the number of new predicates generated to avoid code duplication.

3.7 RELATED WORK

Debray [44] and Sehr [123, 124] use control flow graphs to optimize sequential programs and extract parallelism, respectively. However, both authors considered only intraprocedural control flow. In our formulation, procedure calls disappear in an interprocedural sea of assignments, continuation creation and primitive operations.

Debray et al have recently implemented a control flow analysis based on context free grammars [47] in the `jc` system. The net result is equivalent to the success control flow analysis described in this paper, though their analysis also handles concurrency.

Shivers [130] proposed control flow analysis for the purpose of recovering the control flow graph of higher-order functional programs. We have studied control flow analysis for a language with less general and somewhat different control structures. We present an efficient algorithm for computing the possible control flow, an efficient representation for the control information,

measurements on a range of benchmarks and a construction of control flow graphs from the inferred control flow information.

3.8 CONCLUSION

We have developed a translation of Prolog into control flow graphs, by approximating the return sites of all predicates in a program. The approximation is done by solving simple set constraints, and can be reduced into a fast graph algorithm and memoized dictionary lookup. An algorithm to construct a control flow graph using the control flow analysis was given.

The control information thus found can be efficiently represented by considering the structure of the solution of the control flow problem: by classifying the components of the solution, control information can be wholly or partially shared between predicates.

In order to represent this information compactly, we devised a method for sharing control information between predicates. This scheme reduced control information substantially, by a factor of two or more for 11 of 26 programs and by 50% or more for 7 of the remaining 15 programs.

We close with noting that the choice of indexing algorithm influenced these numbers heavily at times. More sophisticated indexing algorithms, e.g., Refs. [29, 154], could be used to further improve the precision of the control flow analysis.

Future work. Future work involves extending the analysis to handle exceptions and dynamic scheduling, improving the precision of the CFA, developing optimizations based on the ideas developed in this paper [94], and using the CFG for native code compilation. We expect previously developed techniques [154, 142] to be useful in the latter regard.

Acknowledgements. I would like to thank Per Mildner and Håkan Millroth for valuable discussions and their comments on this paper, and Saumya Debray for valuable comments. I am grateful to the anonymous referees for suggestions that improved the presentation and contents of this paper.

3.9 APPENDIX: EXAMPLE OF CONTROL FLOW CONSTRAINTS

Consider the Takeuchi program, with locations for labels shown as comments.

```
tak :- tak(18,12,6,_X).
tak(X,Y,Z,A) :-
```

```
( X =< Y, !, Z = A
; % label(4)
  X > Y,
  X1 is X - 1,
  tak(X1,Y,Z,A1),  % label(1)
  Y1 is Y - 1,
  tak(Y1,Z,X,A2),  % label(2)
  Z1 is Z - 1,
  tak(Z1,X,Y,A3),  % label(3)
  tak(A1,A2,A3,A)
).
```

The following constraints are generated. We assume $ denotes exit with success and # exit with failure.

succ($tak/0$)	$\supseteq$	$\{\$\}$
fail($tak/0$)	$\supseteq$	$\{\#\}$
redo($tak/0$)	$\supseteq$	redo($tak/4$)
succ($tak/4$)	$\supseteq$	succ($tak/0$) $\cup \{1, 2, 3\}$
fail($tak/4$)	$\supseteq$	redo($tak/4$)
redo($tak/4$)	$\supseteq$	fail($tak/0$)

Note that the cut prevents label 4 from appearing for deep backtracking.

3.10 APPENDIX: CLASSES OF STRONGLY CONNECTED COMPONENTS

We measured how frequently the proposed classes of strongly connected components occurred in our set of benchmarks.

Program	Σ	triv	jump	coll	if-then	switch-dflt	other
boyer	162	88	6	33	22	1	12
browse	50	21	12	11	4	0	2
chat	1003	688	71	139	30	11	64
crypt	48	36	3	3	1	2	3
div10	14	6	2	5	0	1	0
fastmu	34	16	6	10	0	0	2
flatten	132	56	20	30	16	3	7
log10	14	6	2	5	0	1	0
metaqs	45	24	3	8	5	0	5
mu	35	13	3	11	5	0	3
nand	473	310	36	75	20	3	29
nrev	16	5	2	6	3	0	0
ops8	14	6	2	5	0	1	0
poly10	42	20	4	6	4	4	4
prover	68	35	5	15	2	2	9
qsort	15	6	3	4	1	1	0
queens8	34	9	5	14	3	0	3
query	69	56	6	4	1	1	1
reducer	260	147	23	42	15	10	23
sdda	186	91	32	27	11	3	22
serial	36	15	2	7	8	0	4
simple	284	127	33	72	21	1	30
tak	13	5	0	3	2	1	2
times10	14	6	2	5	0	1	0
unify	148	75	23	28	6	3	13
zebra	43	23	6	6	4	0	4

Figure 3.4: Categories of strongly connected components among constraints.

Polyvariant Detection of Uninitialized Arguments of Prolog Predicates

Thomas Lindgren

In Journal of Logic Programming, Vol. 28(3) pp. 217-229, September 1996.

Uninitialized variables are important to high performance Prolog implementations, since they can be bound much more efficiently than standard variables and may reduce the size of environments. In this paper, we propose a straightforward program transformation that detects uninitialized arguments to calls and rewrites the program to make such arguments obvious to the compiler.

Our algorithm detects more uninitialized arguments than previously proposed methods, is robust when declarations are lacking and calling modes vary, and never performs worse than the monovariant method previously described in the literature.

On a substantial set of benchmarks, our algorithm always performs as well as previous methods, and sometimes considerably better. The transformation adds specialized predicates to the program; on the order of 20% of the original number of predicates.

4.1 INTRODUCTION

A frequent runtime operation in Prolog is variable binding. Unfortunately, the general case in a common implementation such as the WAM [159] is quite expensive: a pointer chain of unknown length (zero steps or longer)

must be followed, the 'age' of the variable tested, the address of the variable pushed on a stack if the variable is too old, and finally the new value is stored in the cell.

An important special case is the *uninitialized variable*, first studied by Beer [14] and further explored by Van Roy [154], Getzinger [58, 57] and Bigot, Gudeman and Debray [22]. Roughly speaking, a variable is uninitialized if it is unaliased and appears the first time in the current goal.

An uninitialized variable corresponds to an output argument of a predicate call, and the binding of an uninitialized variable can be reduced to assignment (either of a register or a memory location). No dereferencing is needed, since the variable is unaliased. No trailing is needed, since the variable is an output argument of the predicate. In some cases, the binding can even be returned in a register [154, 22], even though Prolog normally returns results in memory.

Beer's approach was to tag uninitialized variables as such and dynamically test for uninitializedness; Van Roy instead detected them through global analysis. Debray et al relied on declared outputs in the framework of a concurrent logic language.

In this paper, we develop a simple syntactic transformation that detects more uninitialized variables than previous methods. The transformation does not rely on declarations, is quite straightforward and has been fully implemented.

The structure of the rest of the paper is the following. Sections 2 and 3 describe the polyvariant transformation; Section 4 gives some concrete examples of how the algorithm works; Section 5 presents an empirical evaluation of the transformation; Section 6 discusses related work, and Section 7 concludes the paper.

4.2 TRANSFORMING PREDICATES

Transforming a goal.

Take a procedure call $P = p(t_1, \ldots, t_k)$, and a set of previously initialized variables V. We compute the set U of uninitialized arguments as follows. (Note that $V \cap U = \emptyset$ up to the last step.) Intuitively speaking, an argument to a call is uninitialized if the actual argument is a variable that occurs the first time in the present call, and does not occur twice in the call.

- Set $U = \emptyset$.
- For each argument t of P:
    1. If $t \in V \cup U$ then $V := V \cup \{t\}; U := U \backslash \{t\}$.

2. If t is a variable and the previous condition does not hold, then $U := U \cup \{t\}$.
3. Otherwise, $V := V \cup \text{fv}(t)$. (The function fv returns the free variables of t.)

The call arguments that appear in U are uninitialized; they have not appeared previously and do not appear inside terms nor twice in the present call.

- After transforming the goal, set $V := V \cup U$.

For a unification $X = T$, if X is uninitialized then the unification can be rewritten into an 'assignment' $X := T$. Furthermore, if this is the last occurrence of X, then variables in T that appear once and for the first time in the unification are viewed as uninitialized (and not added to V).

This is possible since X was the last reference to T and now is dead: each eligible variable $Y \in \text{fv}(T)$ is at this point the sole reference to the corresponding variable cell, up to the point where we return to the caller. When the predicate returns, we assume that all uninitialized variables have been initialized. This is similar to Van Roy's or Getzinger's treatments [154, 57].

Note that last-use information is vital. Consider the following example.

```
p :- X = [A|As], q(X), r(A,As).

q([c,d]).

r(a,[b]).
```

Since X, A and As appear the first time in `X=[A|As]`, it is tempting to consider A and As uninitialized. However, it is not the last use of X, and thus would not be conservative (an aggressive compiler could optimize bindings of A and As in r/2 into assignments). Instead, A and As must be considered initialized.

We also generate specialized versions of other primitives; the details are straightforward and irrelevant to this paper.

Transforming a clause body.

Using the goal transformation we can formulate the translation of a predicate. We rewrite the predicate into single-clausal form, i.e, turn it into a single clause $H \leftarrow B$ where H is a clause head with a linear sequence of variables as arguments and B is a disjunction of the original clauses. We then

annotate the program with last-use information to handle unifications correctly: the last syntactic occurrence of each program variable is associated with the goal it occurs in.

We assume $H = p(X_1, \ldots, X_m, U_1, \ldots, U_n)$ where $U = \{U_1, \ldots, U_n\}$ are uninitialized arguments.

Take $V_0 = \{X_1, \ldots, X_m\}$ as the set of initialized variables. We transform a clause body as follows. We assume for notational convenience that goals are transformed 'in place' and pass around the set V of variables that have appeared so far. The transformation starts with $V = V_0$.

- For a conjunction (A, B), transform A, which yields V'; then translate B using V' yielding V'', and return V''.
- For a disjunction $(A; B)$, transform A yielding V' and B yielding V''. Return $V' \cup V''$.
- For a goal G, apply the goal transformation described previously.

Other constructs are handled similarly, in particular if-then-else and negation-as-failure.

4.3 TRANSFORMING PROGRAMS

We now turn our local analysis into a global one, by redirecting calls with uninitialized arguments to specialized predicates, while simultaneously generating the required specialized predicates. The centerpiece for this transformation is the *call table*, a memo table that stores calls and the corresponding specialized predicates.

A *call mode* of a predicate is a description of what predicate arguments are uninitialized directly before a call to the predicate. A *success mode* of the same predicate is a description of what predicate arguments are uninitialized directly after a call to the predicate.

We will make the assumption that uninitialized arguments are always initialized by the called procedure; this is reasonable, since garbage collection and implementation becomes generally easier. Likewise, returning uninitialized variables is uncommon, since it implies that the variable is not examined. Since all return arguments are initialized, all success modes are trivial, and we need not keep track of them.

We write the mode of an uninitialized arguments as '$-$' and the mode of an initialized argument as '$+$'. This is slightly different from conventional mode declarations, but we hope it will not cause any confusion. The call mode of a predicate is a tuple of argument modes, e.g., $(+, -, +)$.

We assume that when entry points are given, they are given as proposed for ISO Prolog modules; that is, only as predicate name/arity pairs. No further declarations are required (in particular, no mode declarations for entry points are needed or assumed in the rest of the paper).

Monovariant transformation.

For a monovariant transformation, the call table keeps one call mode per predicate. When the call mode is updated (e.g., a call mode argument changes from '$-$' to '$+$' due to computing a least upper bound), the predicate is reanalyzed[1]. When the analysis is done, we perform the clause transformation for every predicate in the program starting from the stored call modes.

Polyvariant transformation.

A polyvariant analysis is equally simple. The call table T maps tuples of predicate names and call modes to specializations. A specialization consists of a *name* and a *definition* of the specialized predicate. Initially, we seed T with an entry

$$T[p/n, (+, \ldots, +)] := \langle p, d \rangle$$

for every entry point p/n with definition d.

When a predicate p/n is called with call mode M, we check if $\langle p/n, M \rangle$ is in the table; if so, we simply return the name of the specialized version and rewrite the call to invoke the specialized definition instead. If the call mode is *not* found, we invent a new name, enter a new specialization into the table, transform the called predicate and fill in the table entry with the specialized code.

While specializing a clause $H \leftarrow B$ for a mode M, the head H is rewritten to make outputs explicit as in Section 4.2, yielding H'. Then, the body B is rewritten to redirect calls to specializations, yielding B'. Finally, $H' \leftarrow B'$ is returned, and subsequently inserted into the call table.

When the transformation terminates, we extract the program from the table, simply by collecting the definitions stored therein.

Notes.

Since there is no need to reconsider specializations (call modes are never merged) the polyvariant algorithm transforms clause bodies on the fly,

[1] There is no need to reanalyze the callers of the predicate, since success modes never change.

rather than as a post-pass. In the implementation, call modes are stored as bit vectors. This makes table checks inexpensive.

Both the polyvariant and monovariant transformations terminate, since the number of possible call modes is finite (though theoretically exponential in the number of arguments).

In practice, the number of versions seems to be limited as discussed below.

4.4 EXAMPLES

We shall illustrate the workings of our algorithm with two examples.

Assuming that the set of uninitialized arguments of a goal is $U = \{X_1, \ldots, X_n\}$, we will use the notational convenience

$$(X_1, \ldots, X_n) := p(t_1, \ldots, t_m)$$

to suggest that output arguments are in effect assignable. If $n = 1$, we will omit the parentheses, as is conventional. It should be understood that this is equivalent to the original call, if one disregards the knowledge of output arguments.

First example.

Consider the nreverse program without entry point declarations, rewritten into a form reminiscent of Van Roy's Kernel Prolog [154]. A lack of entry points is realistic, e.g., when the program contains call/1 and no indication of what predicates can be called. The polyvariant transformation still manages to break out an efficient sub-program. (Interestingly, it has then also ensured that the new specializations will never be invoked by call/1 and so can be compiled more efficiently.)

```
main :- nreverse([1,2,3,...],X), write(X), nl.

nreverse(A,B) :-
        ( A = [], B = []
        ; A = [X|Xs], nreverse(Xs,Ys), append(Ys,[X],B)
        ).

append(A,B,C) :-
        ( A = [], B = C
        ; A = [X|Xs], C = [X|Zs], append(Xs,B,Zs)
        ).
```

The entry points to the program are main/0, nreverse/2 and append/3.

- Starting with main/0, we find that nreverse is called with mode $(+,-)$ and generate a new version of nreverse/2 with that mode, nreverse_1/2.
- Traversing nreverse_1/2, the algorithm finds that the recursive call has mode $(+,-)$; it is redirected to become nreverse_1/2. Likewise, append/3 is found to have mode $(+,+,-)$. The transformation generates append_1/3.
- When traversing append_1/3, the recursive call is redirected to append_1/3 since the third argument is uninitialized. (Note that we consider Zs to be uninitialized, even though it appears inside a structure.)
- At this point, we are done with main/0. The next entry point is nreverse/2. We find that nreverse/2 invokes itself with mode $(+,-)$, so the recursive call is redirected to nreverse_1/2. The call to append/3 must be retained, however, since the mode is $(+,+,+)$.
- Finally, we consider append/3. Using mode $(+,+,+)$, it cannot be improved: the recursive call is also invoked with $(+,+,+)$.

The new program is as follows.

```
main :- X := nreverse_1([1,2,3,...]), write(X), nl.

B := nreverse_1(A) :-
        ( A = [], B := []
        ; A = [X|Xs], Ys := nreverse_1(Xs), B := append(Ys,[X])
        ).

C := append_1(A,B) :-
        ( A = [], B := C
        ; A = [X|Xs], C := [X|Zs], Zs := append_1(Xs,B)
        ).

nreverse(A,B) :-
        ( A = [], B = []
        ; A = [X|Xs], Ys := nreverse_1(Xs), append(Ys,[X],B)
        ).

append(A,B,C) :-
        ( A = [], B = C
        ; A = [X|Xs], C = [X|Zs], append(Xs,B,Zs)
```

```
        ).
```

If we delete nreverse/2 and append/3, we get precisely the result that Aquarius Prolog would infer if only main/0 was considered an entry point.

Second example.

The second example illustrates when predicates are called in different modes, again without declaring entry points.

```
main :- read(Term), free_vars(Term,FV,[]), write(FV), nl.

free_vars(X,FV0,FV1) :- var(X), !, FV0 = [X|FV1].
free_vars(X,FV0,FV1) :- atomic(X),!, FV0 = FV1.
free_vars(X,FV0,FV1) :-
        X =.. [_|Xs],
        free_vars_list(Xs,FV0,FV1).

free_vars_list([],FV0,FV1) :- FV0 = FV1.
free_vars_list([X|Xs],FV0,FV2) :-
        free_vars(X,FV0,FV1),
        free_vars_list(Xs,FV1,FV2).
```

When the polyvariant transformation is applied, the following versions are generated.

main/0	()
free_vars/3	$(+,+,-)$, $(+,-,-)$, $(+,-,+)$, $(+,+,+)$
free_vars_list/3	$(+,+,-)$,$(+,-,-)$,$(+,-,+)$,$(+,+,+)$

The different versions appear due to the difference list arguments: version $(+,-,+)$ due to giving `[]` as an argument to free_vars/3; version $(+,+,+)$ as a safeguard since we gave no entry declarations, and the other two versions due to the interplay of free_vars/3 and free_vars_list/3.

Note that if main/0 was removed, the versions $(+,-,+)$ and $(+,-,-)$ disappear, but $(+,+,-)$ and $(+,+,+)$ are retained. The $(+,+,-)$ versions appear due to the call site where free_vars_list/3 calls free_vars/3 in mode $(+,+,-)$. This means free_vars/3 in turn calls free_vars_list/3 in mode $(+,+,-)$.

A monovariant transformation only yields versions $(+,+,+)$ for free_vars/3 and free_vars_list/3, even if main/0 is given as entry point. The reason is

that [] appears as an argument to free_vars/3, and that the second argument is threaded through the program — as mentioned previously, we assume procedure calls initialize arguments before returning.

If module entry points are absent, the transformation must assume that any predicate can be called from outside the module. The monovariant transformation then naturally cannot find any uninitialized predicate arguments. (It can still detect local uninitialized arguments to primitives, but such a step does of course not require global information.) In contrast, the polyvariant transformation can extract sub-programs that use uninitialized variables if the program introduces such variables, as shown above.

Being a wily programmer.

Using the observation that calls with uninitialized arguments will generate sub-programs that exploit such arguments, the programmer can ensure that the transformation extracts uninitialized variables by rewriting the intended (if undeclared) entry points as follows.

Consider the predicate:

$$p(X_1, \ldots, X_k, X_{k+1}, \ldots, X_n) \leftarrow B$$

If the module writer intends arguments $\{X_{k+1}, \ldots, X_n\}$ as outputs, the program is simply rewritten into the following:

$$p(X_1, \ldots, X_k, Y_{k+1}, \ldots, Y_n) \leftarrow B, Y_{k+1} = X_{k+1}, \ldots, Y_n = X_n$$

The transformation algorithm now can detect uninitialized outputs in B, subject to the restrictions of our algorithm.

This rewriting makes explicit a transformation also done by the Mercury compiler for strongly typed, strongly moded logic programs [133], i.e., moving the outputs of a call to after that call. One option might be to declare the outputs of module entry points, to get this effect automatically as in Mercury.

If p/n is a simple loop predicate, the above implies one iteration is peeled off the loop, and the rest is spent in the optimised version, as shown above in the nreverse example. (This can be viewed as a simplistic form of call forwarding [23].)

4.5 EMPIRICAL EVALUATION

We have performed the proposed transformation on a set of Prolog benchmark programs. In order to compare the sizes of programs, we listed them on a canonical format using listing/0 (i.e., no blank lines or comments). The number of lines of the resulting file, counted by the UNIX utility `wc`, was taken as the *lines of code* (loc) of the program.

Experiment setup

The Berkeley benchmarks is a well-known suite of Prolog benchmarks used to evaluate Aquarius Prolog. The 24 programs are small to moderate in size (15-900 loc), and total 3400 loc. We also included a collection of five larger benchmarks, in order to study the effects on more substantial programs: plwam (2200 loc), chat (3900 loc), bamspec (1100 loc), kish_fo (2700 loc), sym1_fo (2000 loc). The last two programs use call/1, which was transformed into a first-order dispatching predicate prior to analysis (i.e., the first-order call/1 tests the functor of the argument and then calls the correct goal). Our study does not consider dynamic predicates occurring in the benchmarks.

Note that the program chat occurred in two versions of varying size. One is reported under the Berkeley benchmarks, the other among the second set of programs.

We performed two experiments on the programs. First, we considered the programs with entry points declared. We compared the monovariant transformation with the polyvariant transformation, both as described in Section 4.3.

The second experiment ran the polyvariant algorithm without declaring any entry points. As mentioned above, this mode can be used when the code contains call/1 or similar constructs (in which case we do not know the calling mode), or when the user will not declare entry points.

Our measurements for the monovariant analysis differs from those of Van Roy and Despain [155] (VR below) in the following respects:

- VR work with Horn clauses, which is not required for our algorithm.
- We use the geometric mean, while VR use the arithmetic mean, to present the results.
- Our algorithm eliminates dead procedures, which are not shown in the tables. VR include such procedures in their totals.
- In benchmarks that lacked a `main/0` predicate, we added such a predicate calling the actual benchmark entry point. (This was done to get a uniform benchmark startup convention.)

Otherwise the results for the monovariant analysis appear to be identical to those of Van Roy and Despain.

	preds	vers	U	C	A	C/A	U/A	T/A	C/U
boyer	24	25	17	1	61	0.02	0.28	0.30	0.06
browse	14	14	10	0	42	0.00	0.24	0.24	0.00
chat	155	242	330	83	742	0.11	0.44	0.56	0.25
crypt	9	12	0	3	18	0.17	0.00	0.17	∞
div10	3	3	1	0	3	0.00	0.33	0.33	0.00
fastmu	7	7	11	0	35	0.00	0.31	0.31	0.00
flatten	28	43	22	14	83	0.17	0.27	0.43	0.64
log10	3	3	1	0	3	0.00	0.33	0.33	0.00
meta_qs	8	8	3	0	10	0.00	0.30	0.30	0.00
mu	9	16	2	7	17	0.41	0.12	0.53	3.50
nand	40	47	46	8	174	0.05	0.26	0.31	0.17
nrev	4	4	2	0	5	0.00	0.40	0.40	0.00
ops8	3	3	1	0	3	0.00	0.33	0.33	0.00
poly_10	12	12	9	0	27	0.00	0.33	0.33	0.00
prover	10	10	6	0	22	0.00	0.27	0.27	0.00
qsort	4	4	3	0	7	0.00	0.43	0.43	0.00
queens8	8	8	5	0	16	0.00	0.31	0.31	0.00
query	6	6	6	0	7	0.00	0.86	0.86	0.00
reducer	30	36	17	7	95	0.07	0.18	0.25	0.41
sdda	29	37	17	8	81	0.10	0.21	0.31	0.47
serial	8	9	6	1	16	0.06	0.38	0.44	0.17
simple	67	79	62	13	254	0.05	0.24	0.30	0.21
tak	3	3	1	0	4	0.00	0.25	0.25	0.00
times10	3	3	1	0	3	0.00	0.33	0.33	0.00
unify	28	42	25	18	140	0.13	0.18	0.31	0.72
zebra	6	6	1	0	10	0.00	0.10	0.10	0.00
bamspec	111	135	161	28	498	0.06	0.32	0.38	0.17
chat	413	625	539	214	1601	0.13	0.34	0.47	0.40
kish_fo	218	435	0	283	1099	0.26	0.00	0.26	∞
plwam	220	361	209	132	864	0.15	0.24	0.39	0.63
sym1_fo	51	73	0	23	106	0.22	0.00	0.22	∞

Figure 4.1: Measurements: number of predicates, number of specialized versions, number of always (U) and sometimes (C) uninitialized arguments, total number of arguments (A), ratios between C, U and A. Note: $T = (C + U)$.

Measurements

We measured the number of predicates in the transformed program (counting all specializations of a predicate collectively as one predicate), the number of specializations generated, the number of predicate arguments that always were uninitialized, and the number of arguments that were sometimes uninitialized. The 'always uninitialized' arguments, or unconditionally uninitialized arguments, U correspond to those detected by a monovariant transformation, while the polyvariant transformation also detects the 'sometimes uninitialized', or conditionally uninitialized arguments C. The total number of arguments, uninitialized or not, is written as A.

The full table of measurements can be found as Figure 4.1. The main items of interest are the number of specializations versus the number of original predicates, and the difference between $(C+U)/A$ and U/A.

Precision. From the table, we can see that the majority of uninitialized arguments are often unconditionally uninitialized. For 15 of the programs, there is no difference between the monovariant and the polyvariant transformations; for 16 of the programs, from 2% to 26% of the total number of arguments are conditionally uninitialized.

We take the largest of the benchmark programs, chat and nand of the Berkeley programs and bamspec, chat, kish_fo, plwam and sym1_fo of the other set, as particularly interesting, since they represent a more realistic set of programs. These are called 'the larger programs' in the rest of the paper.

Program	(C+U)/A	U/A	(C+U)/U
chat	0.56	0.45	1.24
nand	0.31	0.26	1.19
bamspec	0.38	0.32	1.19
chat	0.47	0.34	1.38
kish_fo	0.26	0.00	∞
plwam	0.39	0.24	1.62
sym1_fo	0.21	0.00	∞
Geo.mean	0.35	0.31	1.32

We also computed the geometric mean of $(C+U)/U$ for the entire benchmark set, and found it to be 1.14. The geometric mean of the total percentage of uninitialized arguments, $(C+U)/A$ was found to be 0.32 ; one argument in three is detectable as uninitialized for some calls to a predicate, on the average. Counting only unconditional arguments, U/A, yields a geometric mean of 0.28.

Two of the larger programs, kish_fo and sym1_fo, used call/1, and so had no unconditional uninitialized arguments. The algorithm successfully found conditionally uninitialized arguments (e.g., non-call/1 call sites with uninitialized arguments).

For the larger programs, the geometric mean of $(C+U)/A$ and U/A was 0.35 and 0.31, respectively (excluding kish_fo and sym1_fo from the total). The proportion of conditionally uninitialized arguments increased as the programs grew larger, possibly because multiple call sites turned unconditional argument modes into conditional ones.

Number of specializations. We then considered the number of specializations required to generate these results. We computed the geometric mean of the ratio of the number of versions (vers column) and the number of original predicates (preds column), and found it to be 1.18 ; one predicate in six generates an extra version. For the larger programs, the geometric mean was 1.48. Part of the difference is due to the presence of call/1 in kish_fo and sym1_fo.

No declared entry points. We measured the effect of not declaring entry points. In this case, the metric of interest is the number of extra versions generated. The monovariant transformation is unable to detect any uninitialized arguments, since every predicate may be called with any argument.

The precision of the polyvariant transformation lacking entry points is similar to that where entry points are declared. The entry points are still present in the benchmarks, so the polyvariant transformation still extracts that subprogram, with the same results. However, it also considers all other predicates equally interesting as starting points, and so may extract a number of extra specializations. In particular, all the original predicates will be retained with call mode $(+, \ldots, +)$.

We computed the ratio of the number of predicates for the polyvariant transformation without declared entry points to that of the original program, as above. The number of specializations ranged from 1.30 to 2.93 of the number of predicates, with larger programs generally increasing more in size.

The geometric mean for all the programs was 1.64. Two out of three predicates required an extra version without declared entry points. For the larger programs, the geometric mean was 2.06. Every predicate then on the average required one extra version without declared entry points.

Timing results. The polyvariant transformation generally is quite fast. Even without declaring entry points, it rarely took more than a second (excluding preprocessing, garbage collection and I/O time) on a 55 MHz

Sun MP/630 (using 1 processor) under SICStus Prolog 2.1.9. The average execution time over all benchmarks was 405 ms, the median 175 ms. The large version of chat required the most time, 4500 ms. Only four programs required more than 1000 ms.

Summary and discussion.

The polyvariant transformation with entry points is economic, in the sense that it can be computed efficiently, improves on the monovariant transformation (sometimes substantially, e.g., when the program employs call/1, at other times less) and increases the number of predicates modestly.

The number of predicates increases by 64% on the average when no entry points are declared; this is because the program must provide all the original predicates, as well as some specialized versions of these predicates. In general, the increase was greater for the larger programs, which indicates that at least large programs probably require entry point declarations, or compile times may be unacceptably high.

Since most code is written in a modular style, much of the generated code will then be executed rarely or never (unless, e.g., call/1 is used unpredictably). It is there to provide safety for a case that never occurs. We recommend three methods to cope with this problem: ensure that program entry points are declared as far as possible; compress the procedures (e.g., into byte code) and generate native code on demand; or merge versions where benefits are marginal (e.g., using profiling information or estimation of enabled optimizations to weed out unprofitable cases).

A full comparison of the polyvariant and monovariant transformation would entail compiling both of the transformed programs and measuring the differences in execution times as well as actual code size. We have in this study not considered *which* predicates generate extra versions; if those predicates are large, then the actual code size may increase more than the above results show. On the other hand, voluminous programs need not necessarily generate more code; current Prolog native code compilers can reduce code size substantially when mode information is available.

Translation into native code and measuring the executables would reveal whether the larger amount of code also translates into an increase in the size of the executable, and whether there are adverse I-cache effects. Unfortunately, we have at the time of writing no access to a compiler (such as Aquarius Prolog) that enables such a comparison.

4.6 RELATED WORK

Beer [14] was the first to exploit uninitialized variables. This was done by a runtime approach, where registers and heap cells were tagged as uninitial-

ized when created and modified during execution when unifications and similar operations occurred. He found that a large portion of the dynamically occurring variables actually were uninitialized. On a set of benchmarks, he found that dereferencing and trailing could be substantially reduced.

Van Roy [154] defined a static analysis that (among other things) derived uninitialized arguments. Our monovariant transformation in essence mimics the 'uninitializedness' subset of Van Roy's analyzer and achieves approximately the same precision [155]. Our work thus shows that this part of Van Roy's analysis can be factored out of the rest of his analysis without losing precision, and that polyvariance further improves the results while avoiding code explosion.

Getzinger improved on Van Roy's analysis and explored some alternatives, but remained within the monovariant framework [57, 58].

Taylor [142] subsequently incorporated uninitialized variables into his Parma compiler, and reported substantial performance gains. However, no indication of the number of uninitialized arguments derived was given.

The recently developed strongly-typed, strongly-moded logic programming language Mercury [133] restricts programs so that outputs are always uninitialized; however, the compiler requires declarations to this effect.

Bigot, Gudeman and Debray [22] have developed an algorithm to decide which output arguments should be returned in registers, and which should be returned in memory. It may be interesting to consider this for our benchmark set, and to possibly use multiple versions of a predicate for different call sites. The Bigot-Gudeman-Debray algorithm uses a single version per predicate.

Gudeman, de Bosschere, Debray and Kannan [23] have defined call forwarding, as a way to hoist type tests out of loops, or in general when the call site can statically decide tests in the callee. As shown in the nreverse example, our polyvariant transformation occasionally generates crude 'call forwarding' by breaking out calls with uninitialized arguments. This suggests that we could possibly share code between predicate versions. Van Roy's compiler [154] merges multiple calls that have produced the same intermediate code, which can be seen as the reverse of call forwarding.

We are only aware of two implementation of multiple specialization for Prolog programs: Sahlin's partial evaluator Mixtus [117] can generate multiple versions of a predicate, and Puebla and Hermenegildo [113] have recently used multiple specialization to improve the parallelization of &-Prolog.

We finally note that the proposed transformation can be seen as an abstract interpretation [41] followed by a program transformation based on the derived results.

4.7 CONCLUSION

We have proposed a simple polyvariant transformation that detects not only 'always uninitialized' predicate arguments, but also 'sometimes uninitialized' predicate arguments, and exploits such opportunities by specializing calls and predicates.

The transformation does not perform an iterative fix point computation, but instead generates new versions of the called predicate, one per call mode. This method is more robust than summarising all calls, since there is no risk that different calls to the same predicate bleed together to produce inferior modes and thus inferior code.

The transformation can be applied with or without entry point declarations. When entry points are available, the number of extra versions increases modestly – on the order of 20% for our set of benchmark programs. When no entry points are available, a situation occurring when call/1 is used, the number of versions increases with approximately 65% for the benchmark set. In this case, the polyvariant transformation still derives uninitialized arguments, while the monovariant analysis is unable to do so.

Acknowledgements. My thanks go to Per Mildner and Håkan Millroth for reading and commenting on this paper, and to Mats Carlsson for providing the large programs used in benchmarking. My thanks also to the referees for their comments.

A SIMPLE AND EFFICIENT COPYING GARBAGE COLLECTOR FOR PROLOG

Johan Bevemyr and Thomas Lindgren

In *Programming Language Implementation and Logic Programming*, LNCS 844, Springer Verlag, 1994.

We show how to implement efficient copying garbage collection for Prolog. We measure the efficiency of the collector compared to a standard mark-sweep algorithm on several programs. We then show how to accomodate generational garbage collection and Prolog primitives that make the implementation more difficult.

The resulting algorithms are simpler and more efficient than the standard mark-sweep method on a range of benchmarks. The total execution times of the benchmark programs are reduced by 4 to 11 percent.

5.1 INTRODUCTION

Automated storage reclamation for Prolog based on Warren's Abstract Machine (WAM) [159] has several difficulties. Let us consider the architecture of a typical WAM: most data are stored on a global stack (also called the heap), while choice points and environments are stored on a local stack (also referred to as the stack). A trail stack records bindings to be undone on backtracking. We will not consider garbage collection of code space in this paper, atom tables or other miscellaneous areas. There are no pointers from such tables into the garbage collected areas.

The WAM saves the state of the machine whenever a choice point is created. Using this information, stacks can be reset and storage reclaimed cheaply. We can view the global stack as composed of several segments, delimited by

the choice point stack. Creating a new choice point creates a new segment; backtracking removes segments, while performing a cut merges segments. Data are allocated in the topmost segment, while variable bindings, which are implemented as assigning a cell representing the variable, are recorded on the trail stack whenever the variable cell is not in the topmost segment. When two variables are unified, a pointer from one cell to the other cell is created. In general, pointer chains may arise which require dereferencing.

A garbage collector for Prolog might thus retain the segment ordering to enable fast storage reclamation on failure by deallocating a segment allocated after the topmost choice point was created. Furthermore, since there are primitives (such as `@<`/2) that compare variables, e.g., by creation time, most systems elect to preserve the heap ordering of data after garbage collection.

Garbage collection is done by starting at a set of *root pointers*, such as registers and the local stack, and discovering what data are reachable from these pointers, or *live*. Memory is reclaimed by compacting the live data [101], copying them to a new area [30] or putting the dead data on a free list. Memory allocation can then be resumed.

5.2 RELATED WORK

Prolog implementations such as SICStus Prolog use a mark-sweep algorithm that first marks the live data, then compacts the heap. We take the implementation of Appleby et al. [8] as typical. This algorithm works in four steps and is based on the Deutsch-Schorr-Waite [121, 36] algorithm for marking and on Morris' algorithm [101, 36] for compacting.

1. All live data are marked through roots found in registers, choice points, environments, and value trail entries (entries in the trail where the old value have been recorded, e.g., as a result of using `setarg`/3). A live tree is marked using a nonrecursive pointer-reversing algorithm that does not require any extra space to operate.

2. The registers, choice points, environments, and the trail are examined for references into the heap. All such references are put into reallocation chains, with the heap cell as root, to be updated when the heap cell is moved.

3. The heap is scanned upwards and all upward references are put into reallocation chains in order to be updated when the cell they refer to is moved.

4. The heap is scanned downwards and all marked data are moved to their new locations. All references to a moved object are found through

the reallocation chains and updated. All references downward are also put into reallocation chains so that they may be updated when the object further down the heap is moved.

Touati and Hama [149] developed a generational copying garbage collector. The heap is split into an old and a new generation. Their algorithm uses copying when the new generation consists of the top most heap segment, i.e., no choice point is present in the new generation, and no troublesome primitives have been used (primitives that rely on a fixed heap ordering of variables). For the older generation they use a mark-sweep algorithm. The technique is similar to that described by Barklund and Millroth [13] and later by Older and Rummell [107].

We show how a simpler copying collector can be implemented, how the troublesome primitives can be accomodated better and how generational collection can be done in a simple and intuitive way. However, our view is also more radical than theirs. Where Touati and Hama still wish to retain properties such as memory recovery on backtracking, we take a more radical approach: ease of garbage collection is more important than recovering memory on backtracking. We show below that memory recovery by backtracking is still possible, and that the new approach in practice recovers approximately as much garbage by backtracking as the conventional approach.

Bekkers, Ridoux and Ungaro [15] describe an algorithm for copying garbage collector for Prolog. They observe that it is possible to reclaim garbage collected data on backtracking if copying starts at the oldest choice point (bottom-to-top). However, their method has several differences to ours.

- Their algorithm does not preserve the heap order, which means primitives such as `@</2` will work incorrectly. They do not indicate how this problem should be solved.
- Their algorithm (the version that incorporates early reset) copies data twice, while our algorithm visits data once and then copies the visited data. We think our approach leads to better locality of reference. However, we have not found any published measurements of the efficiency of the Bekkers-Ridoux-Ungaro algorithm.
- Variable shunting is used to avoid duplication of variables inside structures. This may introduce new variable chains, as shown in Appendix A. We want to avoid this situation.

Their algorithm does preserve the segment-structure of the heap (but not the ordering within a segment). Hence, they can reclaim all memory by backtracking. In contrast, our algorithm only supports partial reclamation

of memory by backtracking. Our measurements indicate that this is sufficient: the copying algorithms we describe do not reclaim appreciably less memory on backtracking than the standard mark-sweep algorithm on the measured benchmarks.

Appel [3, 4] describes a simple generational garbage collector for Standard ML. The collector uses Cheney's garbage collection algorithm, which is the basis of our algorithm as well. However, his collector relies on assignments being infrequent. In Prolog, variable binding is assignment in this sense. Our algorithm handles frequent assignments efficiently.

Sahlin [115] has developed a method that makes the execution time of the Appleby et al. [8] algorithm proportional to the size of the live data. The main drawback of Sahlin's algorithm is that implementing the mark-sweep algorithm becomes more difficult, not to mention guaranteeing that there are no programming errors in its implementation. To our knowledge it has never been implemented. We also believe that Sahlin's algorithm is not as efficient as ours since it requires an extra pass over the live data, beyond the passes in the Appleby algorithm. Since our algorithm is almost 70 % faster than the Appleby algorithm even when the heap is filled with live data, it is unlikely that Sahlin's algorithm will be more efficient than ours.

5.3 ALGORITHM

We assume the standard term representation of WAM [159]. Our algorithm requires the existence of two tag bits for each cell on the heap, reserved for the use of the garbage collector. These tag bits may either be stored in each cell or in some separate area. One of the bits is used for marking copied cells as forwarded, the other is used for indicating that a cell appears inside a live structure.

Avoiding the heap ordering

Variables in the WAM are represented as self-referring heap cells. The WAM uses the location of a variable for deciding if trailing is required when binding the variable. Hence, variables should not move out of their heap segments. Since our algorithm does not preserve heap segments we must find another solution.

Our solution is simply to trail bindings of variables copied during the last garbage collection. The method to do this efficiently is described in Section "Recovering memory on backtracking". Bindings to the surviving variables from the topmost heap segment will be trailed unnecessarily (as compared to the compacting approach), but other bindings will not be affected. The unnecessary trail entries are deleted by the next collection.

Mark-and-copy

The copying collector is a straightforward adaption of Cheney's algorithm [30] and works in three phases. The algorithm allows the standard optimizations of early reset. The old data reside in `fromspace` and are evacuated into `tospace`.

1. Mark the live data. When a structure is encountered, mark the functor cell and all internal cells. When a simple object is found, mark that cell only.

2. Copy the data using Cheney's breadth-first algorithm. When a marked cell is visited in `fromspace`, do the following:

   (a) Scan backward (towards lower addresses) until an unmarked cell is found.

   (b) Scan forward and evacuate marked cells into `tospace` until an unmarked cell is found. Overwrite the old cells with forwarding pointers to the corresponding cells in the copy.

   Thus, interior pointers are handled correctly. Several adjacent live objects may be evacuated at once. Continue until no cells remain to be evacuated.

3. Update the trail. If a trail entry does not refer to a copied cell (i.e., does not point at a forwarding pointer), it can be deleted. Implementing early reset is done by incorporating this step into the procedure that copies live data from the chain of choice points.

The collector thus visits (and writes) the data once, then writes the copy in `tospace`. We believe locality of reference to be quite good: in the second pass, the marked data will already reside in the cache if the data are sufficiently small.

If we do not mark all cells that occur inside live data structures then duplication of cells could occur. Suppose we have both a reference to a variable in a structure and a reference to the structure, e.g., see Figure 5.1. Suppose we copy the variable before the structure; then we would introduce an extra reference, e.g., see Figure 5.2. This is undesirable since the result of doing garbage collection might then be that more space is required! To solve this, we mark all cells that occur in live data structures. When a marked cell is copied the enclosing structure is also copied by step 2.

Recovering memory on backtracking

A compacting collector preserves the heap segments (see Figure 5.3) and

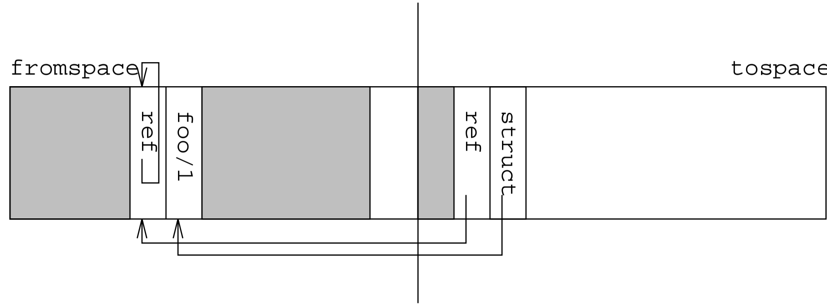


Figure 5.1: An internal cell is referred to twice, both directly by a variable and indirecly through a structure.

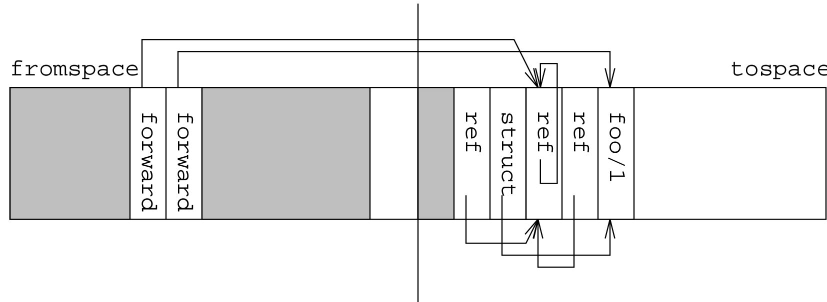


Figure 5.2: As a result of copying without marking internal cells some cells might be duplicated.

entire segments can be deallocated on backtracking. In general, a copying algorithm cannot recover memory on backtracking since the heap no longer preserves the required stack ordering. However, our algorithm can still recover some garbage by resetting the heap pointer, just as in a standard WAM implementation.

We note that *between* collections, memory is allocated just as in a WAM. Using this observation, we can arrange to recover memory allocated after the last collection on backtracking. After a collection, we set the saved heap top of all choice points to the top of the heap space, making this one segment (see Figure 5.4). Terms allocated after this segment can be reclaimed upon backtracking. In this way, we retain most of the advantages of resetting the heap upon backtracking without having to tailor our runtime system and garbage collector to ensure this property at every point. Precisely the same test for trailing a binding can be used as in a standard WAM. Collection

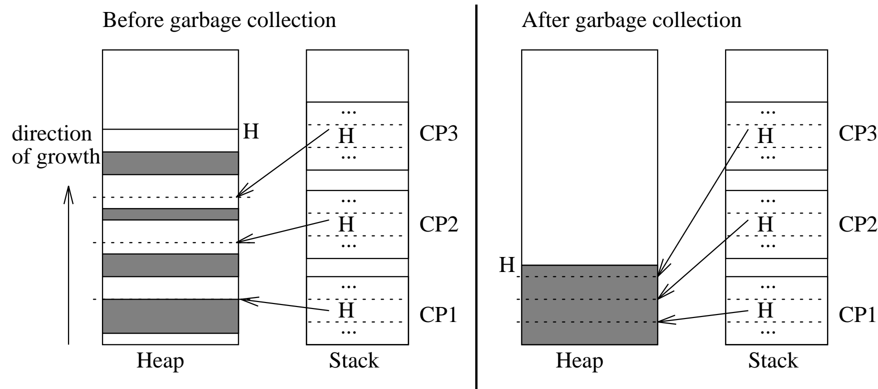


Figure 5.3: Saved heap top pointers (H) in choice points before and after **compacting** garbage collection. Heap segments are preserved.

may also split a single segment into two, which leads to extra trailing. We measure the efficiency of our system below.

Handling troublesome primitives

Touati and Hama recognized that the generic comparison operators, such as `@</2`, required that variables preserve their relative ordering (since variable ordering is usually the comparison criterion when two unbound variables are compared). Their method was either to disable copying collection when these primitives were used and revert to using a compacting collector, or to generate identifiers for variables when needed. The identifiers were to be kept in a hash table to be updated when variables were relocated.

Others have proposed to associate a creation time with each variable. Our experience is that timestamps add a runtime overhead of approximately 5% in an emulator-based WAM implementation.

Our solution retains the use of copying collection, while requiring a small modification to the runtime system. When variables are compared, we arbitrarily order them if unordered. This is done by binding unordered variable cells to new variable cells on a small *compared-variable* stack (cv-stack), see Figure 5.5. Once this is done, we can just compare addresses. The binding is not trailed. (An unbound variable is unordered if it does not reside on the cv-stack and ordered if it is on the cv-stack.)

Once a variable is ordered it resides on the cv-stack. Subsequent unordered variables will be 'greater' than ordered variables when compared, since they are pushed on the cv-stack (where the ordering is kept) when the comparison

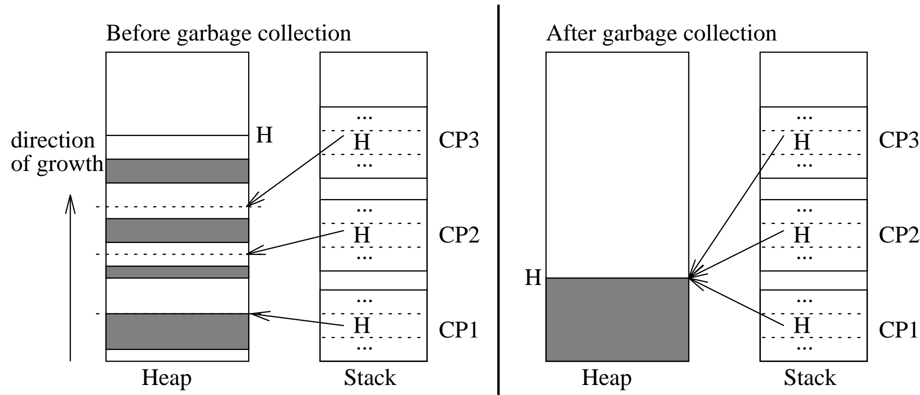


Figure 5.4: Saved heap top pointers (H) in choice points before and after **copying** garbage collection. After garbage collection all segments have been merged into one single segment. Note that only the active heap is shown here—the live data have been copied from the old heap to the new heap.

occurs. Using compacting collection on the cv-stack retains the variable ordering (though the copying collector must take care not to migrate variables residing on the cv-stack), while dead variables disappear.

The space cost is proportional to the number of unbound variables compared by generic comparison operations. Naturally, if most of the live unbound variables have been compared in this way, the collector will have to spend more time in compacting the cv-stack. We believe this situation to be rare.

5.4 INTRODUCING GENERATIONAL GARBAGE COLLECTION

Generational garbage collection [84, 4] relies on the observation that newly created objects tend to be short-lived. Thus, garbage collection should concentrate on recently created data. The heap is split into two or more generations, and the most recent generation is collected most frequently. When the youngest generation fills up, a collection spanning more generations is done, and the survivors move to the oldest of these generations. Frequently, implementations have two generations, and we will assume so from now on.

The difference from standard copying collection is that the collection roots also include the pointers from the older to the younger generation. In languages such as SML, most objects are immutable, and assignments that

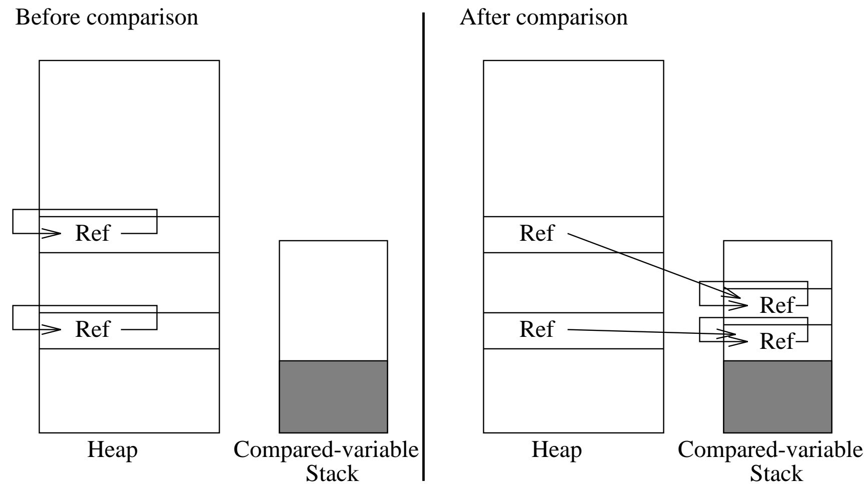


Figure 5.5: Compared variables are bound to variables on a compared-variable stack. This way their relative order is preserved.

may cause cross-generational pointers can be compiled to special code that registers such a pointer if it appears.

In Prolog, there is a high incidence of assigning already created objects, so such a solution is likely to be expensive. Variable bindings involve assignment. In fact, we can arrange so that only *trailed* bindings may be cross-generational, by setting the limit where trailing occurs appropriately, see Figure 5.6. Usually, this limit is the start of the topmost heap segment, but this is not required: we can set it in the interior of the topmost segment at the cost of doing unnecessary trailings. This may be useful if the computation is deterministic but still has a large amount of live data.

We assume that objects are tenured (moved to the old generation) if they survive a collection of the new generation. Now, we can find cross-generation pointers by examining all new trail entries since the last garbage collection. These pointers point out root pointers from the older generation. Since the tenuring threshold is one, the old part of the trail need not be scanned; it can refer only to the old generation. If we were to allow several minor collections before tenuring, the cross generation pointers must be recorded for subsequent use, to avoid scanning the trail repeatedly.

In other languages it is usually necessary to add a *write barrier*, code that detects cross generational bindings and record them on a stack. This result in a runtime cost for using generational garbage collection. In Prolog this

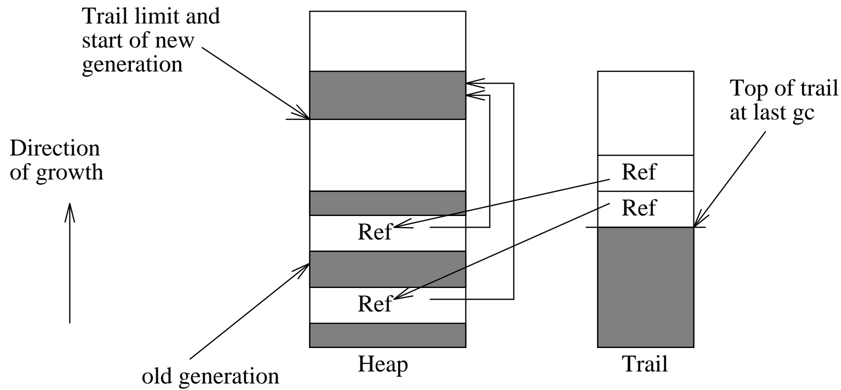


Figure 5.6: The limit for trailing is set in such a way that all cross-generational references are recorded on the trail. In the presence of a choice point in the new generation the trail limit is set as usual.

overhead is already present in the form of trail tests and there is no extra runtime penalty for using generational collection.

5.5 EVALUATION

We have implemented a standard mark-sweep algorithm [8] and compared it to our copying algorithms. All garbage collection algorithms have been implemented in the same system, a sequential version of Reform Prolog. All algorithms implement early reset.

The TSP program implements an approximation algorithm for the Travelling Salesman Problem. A tour of 60 cities was computed. The MATCH program implements a dynamic programming algorithm for comparing, e.g., DNA-sequences. One sequence of length 32 was compared to 100 other sequences. BOYER is the Boyer-Moore theorem prover adapted by Evan Tick to Prolog. This program is part of the Berkeley benchmark suite.

BIG is a program that allocates a large data structure and then forces garbage collection 1000 times while the data structure is still live. The program is intended to compare the copying collectors with mark-sweep when the heap is filled with live data, i.e., when copying and compaction are working on a similar amount of memory and data.

When executing the TSP, MATCH, and BOYER programs we gave each version of the emulator (mark-sweep and copy) an equal amount of memory. This meant that the emulator using copying garbage collection used a heap that was half the size of the mark-sweep heap. One might argue that since

modern computers have virtual memory it is reasonable to let the copy version temporarily allocate twice as much space as the mark-sweep version. However, we have found that even the size of the virtual memory (actually the size of the swap space) can be a limiting factor.

In our measurements of the generational collector, the new generation is given half the remaining free memory in the semi-space, after a minor collection. When the size of the new generation is less than 20K, a major collection is done.

When we executed the BIG program we gave the copying collectors twice the memory of the mark-sweep collector so that the collected data structures could have the same size. The from-space and mark-sweep heap thus were of the same size. This was done in order to compare the timing of a single collection, rather than the entire execution.

All times are in seconds user time.

Program	Heap	Mark-Sweep		Copy		Gen. Copy	
	size	gc	run	gc	run	gc	run
MATCH	750K	12.0%	22.02	2.6%	20.01	0.6%	19.82
	1500K	11.2%	21.89	1.4%	19.70	0.3%	19.64
	2250K	10.4%	21.43	0.7%	19.96	0.1%	19.87
	3000K	10.0%	21.64	0.6%	19.89	0.05%	19.61
TSP	750K	5.3%	54.73	0.9%	51.99	0.1%	51.11
	1500K	4.8%	54.62	0.4%	52.12	0.1%	51.81
	2250K	4.7%	53.75	0.3%	51.96	0.08%	51.48
	3000K	4.6%	55.38	0.2%	51.77	0.07%	51.65
BOYER	750K	15.8%	4.61	15.7%	4.51	8.4%	4.38
	1500K	8.7%	4.25	7.1%	4.23	5.6%	4.09
	2250K	12.4%	4.58	5.9%	4.04	4.9%	4.06
	3000K	0%	4.04	3.0%	3.98	4.9%	4.02
BIG	1500K	100%	11.22	100%	6.34	—	—

The next table shows the difference in execution times for the three algorithms (using 1500K memory). The improvement of the total execution time when using a copying collector ranges from 4 to 11 percent.

We also measured how many times the different garbage collectors were invoked.

Program	Heap size	Number of garbage collections performed		
		Mark-Sweep	Copy	Gen. Copy
MATCH	750K	25	51	130
	1500K	12	25	48
	2250K	8	16	30
	3000K	6	12	22
TSP	750K	22	46	113
	1500K	11	22	51
	2250K	7	14	33
	3000K	5	11	25
BOYER	750K	3	8	30
	1500K	1	3	7
	2250K	1	2	4
	3000K	0	0	3

All times are in milliseconds user time.

	GC time/run time		
Program	Mark-Sweep	Copy	Gen. Copy
MATCH	2460/21890	280/19700	50/19640
TSP	2600/54620	220/52120	50/51810
BOYER	370/4250	300/4230	230/4090

Note that the total execution times are sometimes shorter when garbage collection is performed. We believe this to be due to improved data locality due to copying.

Program	Memory allocated	Memory reclaimed on backtracking		
		Mark-Sweep	Copy	Gen. Copy
MATCH	17770K	793 (4%)	793 (4%)	793 (4%)
TSP	18895K	0 (0%)	0 (0%)	0 (0%)
BOYER	4187K	1798 (43%)	1798 (43%)	1798 (43%)

Approximately the same amount of data is reclaimed on backtracking with all three algorithms. We believe the reason for this is that most of the memory is reclaimed during shallow backtracking.

We also measured the amount of extra trailing imposed by the copying collectors.

Program	Number of trail entries			Ratio	
	Mark-Sweep	Copy	Gen.	Copy/Mark	Gen/Mark
MATCH	112582	112767	112864	1.0016	1.0025
TSP	10560	10568	10568	1.0008	1.0008
BOYER	108352	108450	108441	1.0009	1.0008

Clearly, the extra trailing performed by the copying algorithms is insignificant compared with the total amount of trailing in our benchmark programs. The above measurements were made using a heap size of 750K bytes.

5.6 CONCLUSION

We have described a method for adapting conventional copying garbage collection to Prolog and how to add generational collection to this algorithm. Three problems have been solved, leading to efficient copying and generational copying collectors.

The first problem is interior pointers, which can lead to duplication of data if copied naively. Our method correctly handles interior pointers by marking, then copying data.

The second problem is that copying collection does not preserve the heap ordering. In theory, this means memory cannot be reclaimed by backtracking, and that bindings in the copied area must always be trailed (rather than occasionally).

Our collector exploits that data allocated since the last collection still retain the desired heap ordering. Hence, memory allocated after the last collection can still be reclaimed by backtracking. Our measurements show that our copying algorithm recovers as much memory by backtracking as a conventional ("perfect") mark-sweep algorithm on a range of realistic benchmarks.

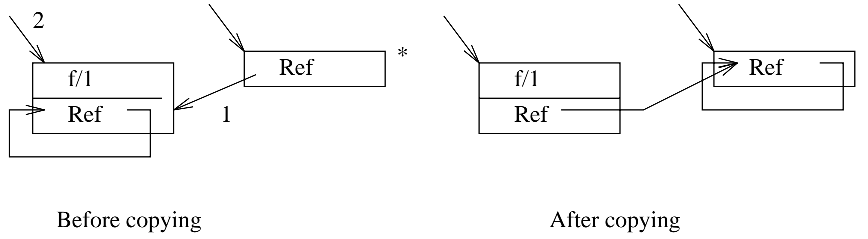
We have also measured the amount of extra trailing due to losing the order of the heap. This was negligible: less than one-quarter of a percent of the total number of trailings at most. We conclude that copying collection is a viable alternative to the conventional mark-sweep algorithm for Prolog.

Finally, we also showed how to extend the copying algorithm to generational collection. The crucial insight is that pointers from the old generation (in a two-generation system) can be found by scanning the trail. By adapting the trailing mechanism, we get an almost-free write-barrier. The only extra cost is some unnecessary trailings in certain situations. This cost is again negligible for our benchmarks.

5.7 ACKNOWLEDGMENT

We thank Ulrich Neumerkel for his comments on this paper, in particular on whether our method to deal with troublesome primitives was correct. We also thank Mikael Pettersson and the anonymous referees for their comments.

5.8 APPENDIX: NEW REFERENCE CHAINS MAY APPEAR FROM VARIABLE SHUNTING



Variable shunting collapses a chain of pointers into a single cell when they are all in the same segment. Assume that pointer (1) in the figure is copied first. The chain of two pointers is collapsed into a single cell $*$ due to variable shunting. Subsequently, reference (2) copies the structure, which yields the situation shown after copying. The number of cells remains constant, but a new reference chain has appeared.

EXPLOITING RECURSION-PARALLELISM IN PROLOG

Johan Bevemyr, Thomas Lindgren and Håkan Millroth

In *PARLE'93*, LNCS 694, Springer Verlag, 1993.

We exploit parallelism across recursion levels in a deterministic subset of Prolog. The implementation extends a conventional Prolog machine with support for data sharing and process management. Extensive global dataflow analysis is employed to facilitate parallelization. Promising performance figures, showing high parallel efficiency and low overhead for parallelization, have been obtained on a 24 processor shared-memory multiprocessor.

6.1 INTRODUCTION

The Single Program Multiple Data (SPMD) model of parallel computation has recently received a lot of attention (see e.g. the article by Bell [16]). The model is characterized by the feature of each parallel process running the same program but with different data.[1] The attraction of this model is that it does not require a dynamic network of parallel processes: this facilitates efficient implementation and makes the parallel control-flow comprehensible for the programmer.

We are concerned here with the SPMD model in the context of logic programming. For recursive programs, the different recursive invocations of a predicate are all executed in parallel, while all other calls are executed sequentially. We refer to this variant of (dependent) AND-parallelism as

[1]This should not be confused with the Single Instruction Multiple Data or SIMD model, where processes are synchronized instruction by instruction.

recursion-parallelism. A recursive invocation minus the head unification and the (next) recursive call is referred to as a *recursion level*. Each recursion level constitutes a process, which gives the programmer an easy way of estimating the process granularity of a given program or call.

We implement recursion-parallelism by *Reform compilation* [100] (this can be viewed as an implementation technique for the Reform inference system [151]). This is a control-structure transformation that changes the control-flow of a recursive program quite dramatically. When invoking a recursive program with a call of size n (corresponding to a recursion depth n) a four-phase computation is initiated:

1. A big head unification, corresponding to the n small head unifications with normal control-flow, is performed.
2. All n instances of the calls to the *left* of the recursive call are computed in parallel.
3. The program is called recursively. This call is known to match the base clause. Hence, in practice, this call is often trivially cheap.
4. All n instances of the calls to the *right* of the recursive call are computed in parallel.

The difference between standard AND-parallelism and recursion-parallelism is illustrated in Figure 6.1. The figure shows execution of a recursive clause $H \leftarrow L, H', R$, where H is the head, H' is the recursive call and L, R are (possibly empty) conjunctions. Note that the figure shows a situation where there are no data dependencies between recursion levels and no unification parallelism.

This paper is organized as follows. In Section 6.2 we define Reform Prolog. We discuss some programming techniques and concepts in Section 6.3. An overview of the parallel abstract machine is presented in Section 6.4. Section 6.5 provides an overview of the compile-time analyses employed for parallelization. Extensions to the sequential instruction set are presented by means of an example in Section 6.6. Experimental results obtained when running benchmark programs on a parallel machine are presented and discussed in Section 6.7.

6.2 REFORM PROLOG

Reform Prolog parallelizes a deterministic subset of Prolog. This is similar to the approach taken in Parallel NU-Prolog [103]. However, Reform Prolog exploits recursion-parallelism when parallelizing this subset, whereas Parallel NU-Prolog exploits AND-parallelism.

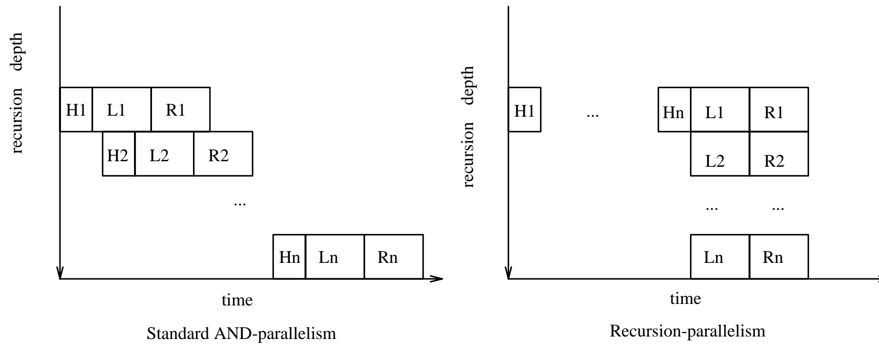


Figure 6.1: Executing a clause $H \leftarrow L, H', R$ with standard AND-parallelism and recursion-parallelism.

With Reform Prolog, as with Parallel NU-Prolog, it is straightforward to call parallel subprograms from a nondeterministic program. Thus, there is a natural embedding of parallel code in ordinary sequential Prolog programs. Since the entire call tree below a parallel call is not parallelized, some non-deterministic computations can be supported in a parallel context as shown below. This is not done in Parallel NU-Prolog.

Below we define the condition for when a recursive Prolog predicate can be parallelized, and how this condition can be enforced by the implementation. We need to define two auxiliary concepts:

- A variable is *shared* if it is visible from more than one recursion level. Note that a variable can be shared at one point of time and unshared (local) at another.
- A variable binding is *unconditional* if it cannot be undone by back-tracking.

We say that a call in a parallel computation is *safe* if all bindings made to its shared variables are unconditional when the call is executed. The condition for when a predicate can be parallelized is then:

> **A recursive predicate can be parallelized only if all calls made in the parallel computation are safe**.

Hence, limited nondeterminism is allowed: when conditional bindings are made only to variables local to a recursion level, the computation is safe.

Safeness of a call is defined w.r.t. to the *instantiation* of the call (i.e., what parts of the arguments are instantiated). We can distinguish between

the parallel instantiation and the sequential instantiation of a call. These might differ as a parallel call can 'run ahead' of the sequential instantiation: recursion levels that would execute after the current one sequentially, may already have bound shared variables.

We say that a call is *par-safe* when it is safe w.r.t. the parallel instantiation, and that it is *seq-safe* when it is safe w.r.t. the sequential instantiation.

The compiler is responsible for checking that programs declared parallel by the programmer are safe. For calls that can be proven par-safe at compile-time, there is no need for extra safeness checking at runtime. For calls that can be proven seq-safe at compile-time, but not par-safe, it is necessary to check safeness at runtime. If the call is not safe, then it is delayed until it becomes safe. This is done by suspending the call until:

1. The unsafe argument has become sufficiently instantiated by another recursion level; or

2. The current call becomes leftmost.

If neither par-safeness nor seq-safeness can be proven at compile-time, then parallelization fails.

6.3 RECURSION PARALLEL PROGRAMMING

Before describing the execution machinery, we briefly consider some programming techniques and concepts.

Machine utilization

The parallel programmer is concerned with utilizing the available parallel machine as efficiently as possible. In Reform Prolog, this means creating sufficient work and avoiding synchronization due to data dependences. A parallel machine where most of the workers are inactive due to lack of work is underutilized; a parallel machine where most workers are waiting for data is inefficiently used.

The number of processes available to the workers is precisely the number of recursion levels of the parallel call. To keep a large machine busy, there should consequently be many recursion levels – far more than the number of workers, ideally.

Safeness

To illustrate the concept of safeness, consider the following two programs:

```
split([],_,[],[]).
split([X|Xs],N,[X|Ys],Zs) :- X =< N, split(Xs,N,Ys,Zs).
split([X|Xs],N,Ys,[X|Zs]) :- X > N, split(Xs,N,Ys,Zs).

split*([],_,[],[]).
split*([X|Xs],N,Ys,Zs) :- X =< N, !, Ys = [X|Ys0], split*(Xs,N,Ys0,Zs).
split*([X|Xs],N,Ys,[X|Zs]) :- split*(Xs,N,Ys,Zs).
```

Assume that the third arguments of both predicates are shared, as might be the case if they were called from a parallel predicate. The predicate split/4 is then unsafe, since the third argument might be conditionally bound in the second clause, and then unbound again if the comparison $X \leq N$ fails. In contrast, the third argument of split∗/4 is only bound in a determinate state and so split∗/4 is safe for parallel execution (the binding of the fourth argument in the last clause does not affect safeness, since clauses are tried in textual order).

Suspension

The programmer would like to avoid suspension as far as possible. However, in the implementation described in this paper, cheap suspension and simple, efficient scheduling combine to lessen synchronization penalties considerably. Consider the following program:

```
nrev([],[]).
nrev([X|Xs],Zs) :- nrev(Xs,Ys), append(Ys,[X],Zs).

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

The compiler detects that the first argument of append/3 is shared. Hence, indexing must suspend until the first argument Ys is instantiated by the previous recursion level. The third argument Zs is also found to be shared, but the situation is reversed: the next recursion level will wait for Zs to be bound.

The inner loop of append/3 is then: wait for the input to be instantiated; when this occurs, write an element on the output list and go back to the beginning again. Thus, there is considerable scope for overlapping computations. As can be seen in the benchmark section, speedups are almost linear on 24 processors.

We also ran a second version of nrev/2, where the data dependence of append/3 was removed by a simple transformation: the length of the first argument is known at call-time by an extra parameter. Every call to append/3 can then construct the list Ys asynchronously (the elements of the list will be filled in later) and there is no suspension under execution.

```
nrev*(0,[],[]).
nrev*(N+1,[X|Xs],Zs) :- nrev*(N,Xs,Ys), append*(N,Ys,[X],Zs).

append*(0,[],Ys,Ys).
append*(N+1,[X|Xs],Ys,[X|Zs]) :- append*(N,Xs,Ys,Zs).
```

Surprisingly, the nrev∗/3 program is slower than the suspending nrev/2 program. Measurements show that this is because recursion levels of nrev/2 usually suspend very briefly, due to simple, fast suspension and straightforward scheduling of processes. For 16 processors, no processor was suspended more than 0.6% of the total execution time on the nrev/2 program; when run on 8 processors, the program suspended each worker less than 0.1% of the execution time.

On the other hand, the asynchronous nature of constructing the answer lists in nrev∗/3 led to an increase in the number of general unifications, due to later recursion levels overtaking earlier ones. The cost is time and memory. (Note that nrev∗/3 still exhibited a speedup of approximately 13 on 16 processors; nrev/2, however, was clocked at a speedup of over 15 on 16 processors.)

6.4 THE PARALLEL ABSTRACT MACHINE: OVERVIEW

The Reform engine consists of a set of *workers*, at least one per processor. Each worker is a separate process running a WAM-based [159] Prolog engine with extensions to support parallel execution. The Prolog engine is comparable in speed with SICStus Prolog (slightly faster on some programs, slightly slower on others).

The Reform engine alternates between two modes: sequential execution and parallel execution. One dedicated worker (the sequential worker) is responsible for the sequential execution phase. During this phase all other workers (the parallel workers) are idle. The sequential worker initiates parallel execution and resumes sequential execution when all parallel workers have terminated.

A common code area is used and all workers have restricted access to each others' heaps. All other data areas are private to each worker. The shared heaps are used to communicate data created during sequential and parallel execution (an atomic exchange operation is employed when binding possibly shared heap variables). When there are several shared heaps it is no longer possible to use a simple pointer comparison for determining whether a binding should be trailed or not. We solve this problem by extending the representation of each variable with a timestamp.

The sequential worker's temporary registers can be read by the parallel workers. These registers are employed for passing arguments to the parallel

computation (one such register contains the precomputed recursion depth, i.e., the total number of parallel processes).

Synchronization is implemented by busy-waiting, i.e., suspended processes actively check if they can be resumed. The drawback of this method is that a suspended process ties up a processor. The advantage is that nonsuspended processes are not slowed down. In particular, very simplisitic and efficient approaches to process scheduling are possible. The Reform Prolog implementation currently supports *static scheduling* and *dynamic self-scheduling* [138]. With both approaches, the actual task switching amounts to a simple jump operation.

6.5 COMPILING RECURSION PARALLEL PROGRAMS

The compiler ensures the safeness of the computation, guarantees that time-dependent operations are performed correctly and employs suspending and locking unification when necessary.

These tasks depend on compile-time program analyses that uncover type, locality and determinacy information. The compiler then emits instructions based on this information, possibly falling back to more conservative code generation schemes when high-precision analysis results cannot be obtained.

Type analysis

The type inference phase employs an abstract domain based on the standard mode-analysis domain [44], augmented with support for lists and difference lists as well as handling of certain aliases.

The compiler distinguishes the parallel and sequential types of a predicate. The sequential type is the type that must hold when the current recursion level is leftmost, while the parallel type holds for any recursion level. The parallel type is the most frequently used for compilation.

Locality analysis

Locality analysis tries to find what terms are local to a process (recursion level), what terms are shared between processes and what terms contain variables subject to time-dependent tests. Consider the following program:

```
rp([],[]).
rp([X|Xs],[Y|Ys]) :- p(X,Y), rp(Xs).

p(a,X) :- q(X).                    p(b,c).

q(X) :- var(X),!,X = b.            q(c).
```

Given the call rp([a,b],[Y,Y]), we get two processes, p(a,Y) and p(b,Y). Both are safe and can thus proceed in parallel. Now assume p(b,Y) precedes p(a,Y), and binds Y to c. Then p(a,c) will reduce to q(c) which succeeds.

Sequential Prolog would have a quite different behaviour: first, p(a,Y) reduces to q(Y) and in turn to Y = b. Then p(b,b) fails. Hence, for this example, the parallel execution model is unsound w.r.t. sequential Prolog execution. This is due to the time-dependent behaviour of the primitive var/1.

Our solution to this problem is to mark, at compile-time, certain variables as being time-sensitive, or *fragile*. In the example, the argument of q/1 is fragile and the compiler must take this into account.

Furthermore, knowledge that a variable is *not* fragile, or *not* shared is extremely useful to the code generator. Using such information, operations with a very complex general case, such as unification, can in some cases be reduced to the same code as would be executed by a sequential WAM.

Safeness analysis

Safeness analysis aims at ensuring that no conditional bindings are made to shared variables. In this respect, it is quite different from determinacy analysis: while determinacy analysis attempts to prove that a given call yields a single solution at most, safeness analysis instead proves no unifications with shared terms are made in a nondeterminate, parallel state. Safeness analysis thus employs the results of type inference (to see whether the call is determinate or not) and locality analysis (only shared terms can be unsafe).

6.6 INSTRUCTION SET

The sequential WAM instruction set is extended with new instructions for supporting recursion-parallelism. Due to space limitations, we can only describe these by means of a simple example and refer to other sources for a full discussion [20, 17, 85]. Consider the program:

```
map([],[]).
map([A|As],[B|Bs]) :- foo(A,B), map(As,Bs).
```

The program is compiled into the following extended WAM code.

```
map/2:  switch_on_term Lv La L1 fail

Lv:     try La
        trust L1
```

```
La:     get_nil X0
        get_nil X1
        proceed

L1:     build_rec_poslist X0 X2 X3 X0     % first list
        build_poslist X1 X2 X4 X1         % second list
        start_left_body L2                % execute L2 in parallel
        execute map/2                     % call base case

L2:     initialize_left 1                 % I := initial recursion level
L3:     spawn_left 1 X2 G2                % X2 := I++; while(I < N) do
        put_nth_head G3 X2 0 X0           %  X0 := G3[I]
        put_nth_head G4 X2 0 X1           %  X1 := G4[I]
        call foo/2 0                      %  call foo(G3[I],G4[I])
        jump L3                           % next iteration
```

The instructions `build_rec_poslist` and `build_poslist` employ a data structure called a *vector-list*. This is a list where the cells are allocated in consecutive memory locations, to allow constant-time indexing into the list. The instructions work as follows:

- `build_rec_poslist X0 X2 X3 X0` traverses the list in X0 and builds a vector-list of its elements, storing a pointer to it in X3, and storing the vector length in X2. The last tail of the list is stored in X0.
- `build_poslist X1 X2 X4 X1` traverses the list X1 to its X2'th element, and builds a vector-list of its elements in X4. If the list has fewer than X2 elements and ends with an unbound variable, then it is filled out to length X2. The X2'th tail of the vector-list is unified with the X2'th tail of the list in X1, and finally X1 is set to the X2'th tail of the vector-list.

The sequential worker's registers X2, X3 and X4 are referred to as G2, G3 and G4 in the parallel code.

The instruction `initialize_left` calculates the initial recursion level in static scheduling mode. In dynamic scheduling mode, this instruction is ignored.

6.7 EXPERIMENTAL RESULTS

In this section we present the results obtained when running some benchmark programs in Reform Prolog on a parallel machine.

Experimental methodology

Parallel machine. Reform Prolog has been implemented on the Sequent Symmetry multiprocessor. This is a bus-based, cache-coherent shared-

memory machine using Intel 80386 processors. The experiments described here were conducted on a machine with 26 processors, where we used 24 processors (leaving two processors for operating systems activitites).

Evaluation metrics. The metric we use for evaluating parallelization is the speedup it yields. We present *relative* and *normalized* speedups.

Relative speedup expresses the ratio of execution time of the program (compiled with parallelization) on a single processor to the time using p processors.

Normalized speedup expresses the ratio of execution time of the original sequential program (compiled without parallelization) on a single processor to the time using p processors on the program compiled with parallelization.

Benchmarks.

Programs and input data. We have parallelized four benchmark programs. Two programs (Match and Tsp) are considerably larger than the other two. One program (Map) exploits independent AND-parallelism, whereas the other three exploit dependent AND-parallelism.

Map. This program applies a function to each element of a list producing a new list. The function merely decrements a counter 100 times. A list of 10,000 elements was used.

Nrev. This program reverses a list using list concatenation ('naive reverse'). A list of 900 elements was used.

Match. This program employs a dynamic programming algorithm for comparing, e.g., DNA-sequences. One sequence of length 32 was compared with 24 other sequences. The resulting similarity-values are collected in a sorted binary tree.

Tsp. This program implements an approximation algorithm for the Travelling Salesman Problem. A tour of 45 cities was computed.

Load balance. One way of estimating the load balance of a computation is to measure the finishing time of the workers. We measured the execution time for each worker when executing our benchmarks. Static scheduling was used in all experiments.

Map. This program displayed a very uniform load balance (less than 0.3% difference between workers). This is hardly surprising since the number of recursion levels executed by each worker is large, and there is no difference in execution time between recursion levels.

Nrev. The execution time of each worker only varied about 3% when executing this program. There is a slight difference in the execution time of each recursion level but the large number of recursion levels executed by each worker evens out the differences.

Match. When 16 workers were used, 8 workers executed 2 recursion levels each, while 8 workers executed a single recursion level. This explains the relatively poor speedup on 16 workers. When 24 workers were used the execution time varied less than 0.3% between workers. This is explained by the fact that each worker executed one recursion level, and that all recursion levels executed in the same time.

Match and Tsp. These program displayed a uniform load balance on all but three workers. This is explained by the fact that 45 recursion levels were executed in all; 21 workers executed 2 recursion levels each while 3 workers executed 1 recursion level each. Despite this the program displays good speedup (21.85). Using dynamic scheduling would not have improved the results in this case.

Sequential fraction of runtime. Each parallelized benchmark program has an initial part which is not parallelized. This includes setting up the arguments for the parallel call. It also includes head unification and spawning of parallel processes.

According to Amdahl's law, the ratio of time spent in this sequential part of the program to that spent in the part which is parallelized (measured on a sequential machine) determines the theoretically best possible speedup from parallelization.

The following table shows for each benchmark program how large a fraction of the execution time on a sequential machine is not subject to parallelization.

Map	Nrev	String	Tsp
0.3%	0.04%	0.003%	0.005%

We conclude from this data that the unparallelized parts represent negligible fractions of the total execution times. Another conclusion is that there is no point in parallelizing the head unification of parallelized predicates, since it represents such a tiny fraction of the computation.

Results

The results of the experiments are summarized in the tables below. In the tables P stands for number of workers, T for runtime (in seconds), S_R for

relative speedup, and S_N for normalized speedup. The sequential runtime for each program is given below each table.

P	T	S_R	S_N	T	S_R	S_N
1	40.40	1.00	0.98	30.80	1.00	0.88
4	10.12	3.99	3.89	8.08	3.81	3.43
8	5.07	7.96	7.76	3.96	7.77	6.99
16	2.54	15.91	15.50	2.01	15.32	13.78
24	1.70	23.76	23.15	1.36	22.65	20.36

Map. (39.59 sec.) **Nrev**. (27.70 sec.)

P	T	S_R	S_N	T	S_R	S_N
1	68.88	1.00	0.95	258.22	1.00	0.90
4	17.22	3.99	3.80	68.85	3.75	3.37
8	8.61	7.99	7.60	34.55	7.47	6.73
16	5.76	11.95	11.35	17.25	14.96	13.47
24	2.91	23.70	22.52	11.82	21.85	19.67

Match. (65.44 sec.) **Tsp**. (232.40 sec.)

Discussion

From the above results we calculate parallel overhead and efficiency of parallelization and make a comparison with other systems.

We compare Reform Prolog with the only other Prolog systems supporting deterministic dependent AND-parallelism that we are aware of, Andorra-I [118] and NUA-Prolog [108]. The Andorra-I system is an interpreter written in C. NUA-Prolog is a compiled system using a WAM-based abstract machine.

It should be noted that these systems to some extent exploit different forms of parallelism. Reform Prolog and NUA-Prolog exploit deterministic dependent AND-parallelism (recursion parallelism in the case of Reform Prolog). Andorra-I exploits deterministic dependent AND-parallelism and OR-parallelism (here we are only interested in the AND-parallel component of the system).

Unfortunately, we can make only a very limited comparison with NUA-Prolog, since the published benchmark programs stress the constraint-solving capabilites of the system, rather than its potential for raw AND-parallel speedup. However, we have compared their results on the nrev benchmarks with ours.

Let us define

$$\text{parallel efficiency} = (\text{speedup on } N \text{ processors})/N$$

The table below displays the parallel efficiency of Reform Prolog on 24 processors. It also indicates the parallelization overhead on a single processor as compared to the sequential Prolog implementation.

Program	Relative efficiency	Normalized efficiency	Parallelization overhead
Map	99 %	96 %	2 %
Nrev	95 %	83 %	12 %
Match	99 %	94 %	5 %
Tsp	91 %	82 %	10 %

The Andorra-I system shows relative efficiency in the range of 47–83 % and normalized efficiency in the range of 35–61 % (assuming parallelization overhead of 35 %). We have excluded benchmarks that mainly exhibits OR-parallelism from this comparison. The figures are obtained on a Sequent Symmetry with 10 processors. NUA-Prolog shows a relative efficiency of 71 % and a normalized efficiency of 36 % on the nrev benchmark on an 11 processor Sequent Symmetry. (Note that the Reform Prolog figures were obtained using more than twice the number of processors the other systems used—using less processors improves the result.)

The Andorra-I system shows parallelization overheads in the range of 35–40 % on a set of benchmarks [118]. NUA-Prolog shows a parallelization overhead of 50 % on the nrev benchmark.

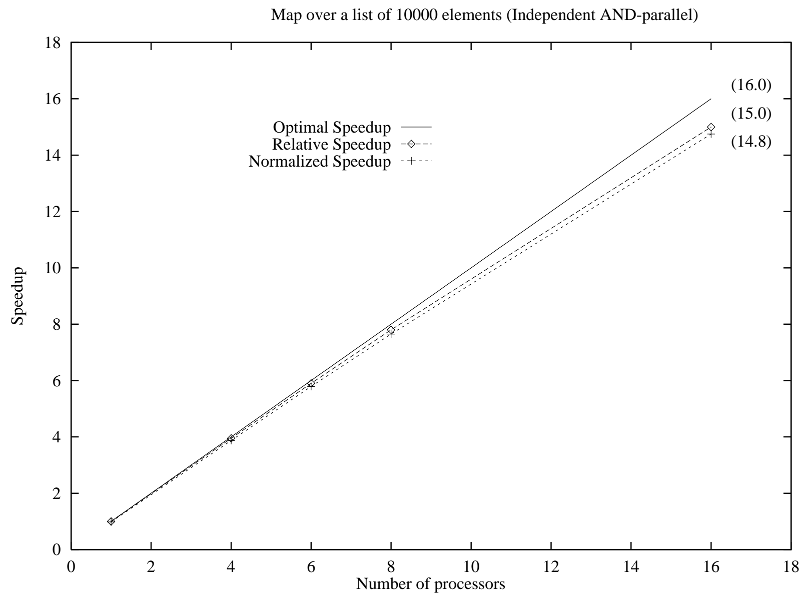
6.8 CONCLUSIONS

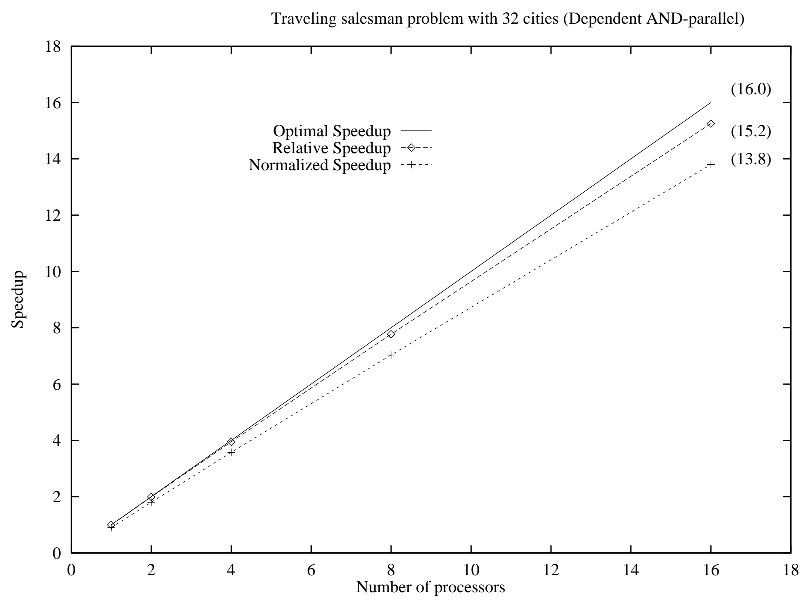
The developments and results discussed in this paper suggest that recursion-parallelism is an efficient method for executing Prolog programs on shared-memory multiprocessors. Our implementation exhibits very low overhead for parallelization (2–12 % on the programs tested).

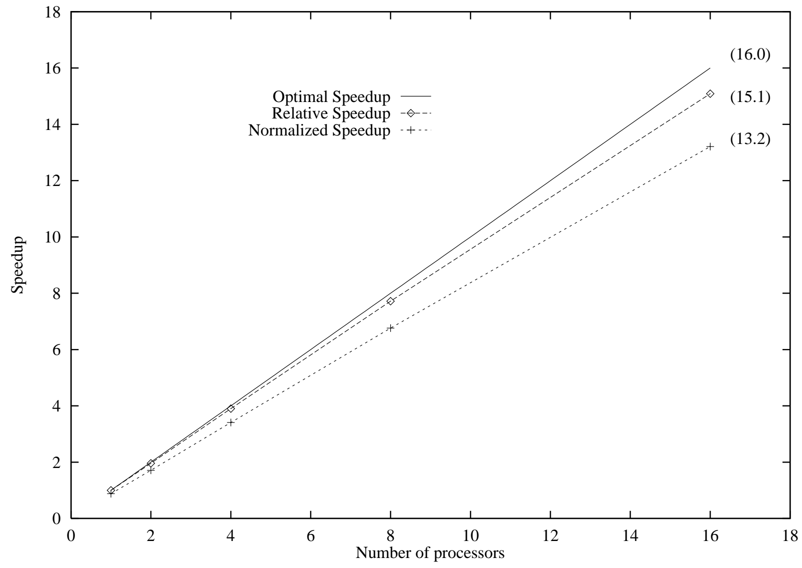
We believe that the high parallel efficiency of Reform Prolog is due to efficient process management and scheduling. An important factor is that all parallel processes can be initiated simultaneously. These properties of the system are due to the static recursion-parallel execution model made possible by Reform compilation.

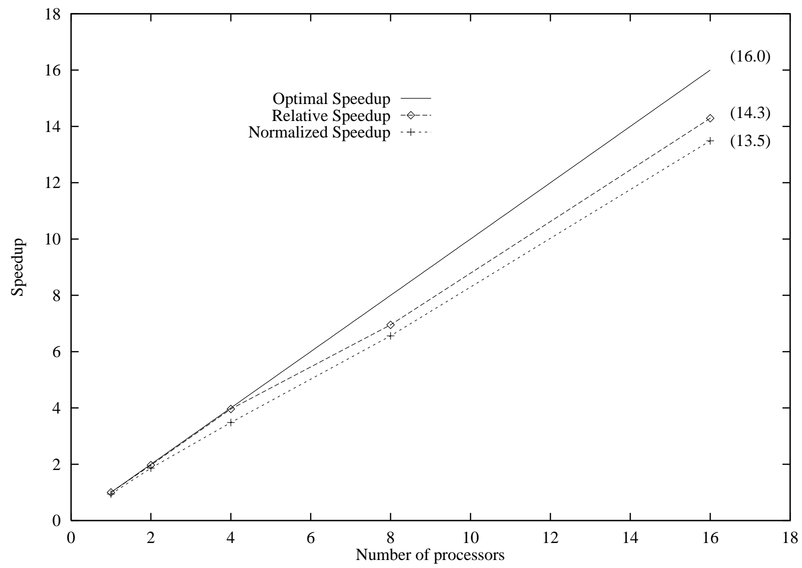
Acknowledgments We thank the Swedish Institute of Computer Science (SICS) for making their 26 processor Sequent Symmetry available to us.

6.9 APPENDIX: BENCHMARK PLOTS






Naive reverse of list with 900 elements (Dependent AND-parallel)
Optimal Speedup
Relative Speedup
Normalized Speedup
(16.0)
(15.1)
(13.2)
Speedup
Number of processors
0
2
4
6
8
10
12
14
16
18


Comparing DNA sequences (Dependent AND-parallel)
Optimal Speedup
Relative Speedup
Normalized Speedup
(16.0)
(14.3)
(13.5)
Speedup
Number of processors
0
2
4
6
8
10
12
14
16
18

Reform Prolog: The Language and its Implementation

Johan Bevemyr, Thomas Lindgren and Håkan Millroth

In *Proc. 10th Int. Conf. Logic programming*, MIT Press, 1993.

Reform Prolog is an (dependent) AND-parallel system based on recursion-parallelism and Reform compilation. The system supports selective, user-declared, parallelization of binding-deterministic Prolog programs (non-determinism local to each parallel process is allowed). The implementation extends a conventional Prolog machine with support for data sharing and process management. Extensive global dataflow analysis is employed to facilitate parallelization. Promising performance figures, showing high parallel efficiency and low overhead for parallelization, have been obtained on a 24 processor shared-memory multiprocessor. The high performance is due to efficient process management and scheduling, made possible by the execution model.

7.1 INTRODUCTION

Most systems for AND-parallel logic programming defines the procedural meaning of conjunction to be inherently parallel. These designs are based on an ambition to maximize the amount of parallelism in computations. We present and evaluate an approach to AND-parallelism aimed at maximizing not parallelism but machine utilization. The system supports selective, user-declared, parallelization of Prolog.

Reform Prolog supports parallelism only across the different recursive invocations of a procedure. Each such invocation constitutes a process, which gives the programmer an easy way of estimating the control-flow and process granularity of a program. We refer to this variant of (dependent) AND-parallelism as *recursion-parallelism*.

We implement recursion-parallelism by *Reform compilation* [100] (this can be viewed as an implementation technique for the Reform inference system [151]). This is a control-structure transformation that changes the control-flow of a recursive program quite dramatically. When invoking a recursive program with a call of size n (corresponding to a recursion depth n) a four-phase computation is initiated:

1. A big head unification, corresponding to the n small head unification with normal control-flow, is performed.
2. All n instances of the calls to the *left* of the recursive call are computed in parallel.
3. The program is called recursively. This call is known to match the base clause. Hence, in practice, this call is often trivially cheap.
4. All n instances of the calls to the *right* of the recursive call are computed in parallel.

This approach is somewhat akin to loop parallelization in imperative languages such as Fortran. However, an important difference is that the granularity of top-level (or near top-level) recursive Prolog procedures typically far exceeds that of parallelizable Fortran loops. A major feature of the approach is that it allows a static process structure: all parallel processes are initialized when the parallel computation starts. This has profound impact on performance.

This paper is organized as follows. The Reform Prolog execution model is defined in Section 7.2. In Sections 7.3 to 7.5, the design of the parallel abstract machine is discussed. The global dataflow analysis employed by the compiler is outlined in Section 7.6. The extension of the instruction set for parallel execution is described in Section 7.7. Section 7.8 presents our experimental results.

7.2 REFORM PROLOG

Reform Prolog parallelizes a deterministic subset of Prolog. Below we define the condition for when a recursive Prolog predicate can be parallelized. We then consider how this condition can be enforced by the implementation. We need to define two auxiliary concepts:

- A variable is *shared* if it is accessible from more than one recursion level. Note that a variable can be shared at one point of time and unshared (local) at another.

- A variable binding is *unconditional* if it cannot be undone by backtracking.

A call in a parallel computation is *safe* if all bindings made to its shared variables are unconditional when the call is executed. The condition for when a predicate can be parallelized is then:

> **A recursive predicate can be parallelized only if all calls made in the parallel computation are safe**.

Safeness of a call is defined w.r.t. to the *instantiation* of the call (i.e., what parts of the arguments are instantiated). We can distinguish between the parallel instantiation and the sequential instantiation of a call. These might differ as a parallel call can 'run ahead' of the sequential instantiation: recursion levels that would execute after the current one sequentially, may already have bound shared variables.

We say that a call is *par-safe* when it is safe w.r.t. the parallel instantiation, and that it is *seq-safe* when it is safe w.r.t. the sequential instantiation.

The compiler is responsible for checking that programs declared parallel by the programmer are safe. For calls that can be proven par-safe at compile-time, there is no need for extra safeness checking at runtime. For calls that can be proven seq-safe at compile-time, but not par-safe, it is necessary to check safeness at runtime. If the call is not safe, then it is delayed until it becomes safe. This is done by suspending until:

1. The unsafe argument has become sufficiently instantiated by another recursion level; or
2. The current call becomes leftmost.

If neither par-safeness nor seq-safeness can be proven at compile-time, parallelization fails.

The execution model described above has some similarities to the approach taken in Parallel NU-Prolog [103] in that both approaches parallelize a binding-deterministic subset of Prolog. However, Reform Prolog exploits recursion-parallelism when parallelizing this subset, whereas Parallel NU-Prolog exploits AND-parallelism.

With Reform Prolog, as with Parallel NU-Prolog, it is straight-forward to call parallel subprograms from a nondeterministic program. Thus, there is a natural embedding of parallel code in ordinary sequential Prolog programs.

Safeness allows local nondeterminism in each recursion level as long as no unsafe bindings are made. In Parallel NU-Prolog this is not done, since

the entire proof tree is subject to parallelization. The consequence is that any variable may be accessible to other processes and so any binding in a nondeterminate state may be unsafe.

7.3 THE PARALLEL ABSTRACT MACHINE

The parallel machinery consists of a set of workers numbered $0,1,\ldots,n-1$, one per processor. Each worker is implemented as a separate process running a WAM-based Prolog engine with extensions to support parallel execution.

Execution phases

The execution of a program alternates between two modes: sequential execution and parallel execution. A phase of sequential execution is referred to as a *sequential phase* and a phase of parallel execution as a *parallel phase*. One worker is responsible for sequential execution (the *sequential* worker). During sequential execution all other workers (the *parallel* workers) are idle, during parallel execution the sequential worker is idle.

Memory Layout

A standard WAM implementation has three main data areas: heap, stack and trail. The heap contains variables and data structures, the stack contains choicepoints and environments, and the trail contains references to bound variables. Environments contain only variables which are either unbound, referring to other variables in the stack, or referring to variables or structures on the heap.

In the Reform engine each worker has its own distinct data areas (heap, stack and trail). The stack and the trail are local to each worker and cannot be accessed by other workers. All heaps are shared and all workers have restricted access to other workers' heaps. The motivation for the design is given below.

Heap. Each parallel worker might require access to data built on the heap of the sequential worker. Moreover, if there are data dependences in the Prolog program, then parts of each workers data might have been created by other workers during the current parallel phase. The easiest way to support this is to let all workers have access to each others heaps. This access is restricted so that no worker may create new objects on another workers heap, but only bind existing variables. Such an arrangement ensures simple memory management of each heap.

Stack. The stacks can remain unshared if it can be ensured that no references to stack objects will occur in shared objects. Since heap variables cannot be bound to stack objects, the only restriction that has to to be imposed is to disallow stack objects in the arguments to the parallel call.

Trail. If each worker is responsible for any unconditional bindings it has created, then the trail can remain local. For this to work, the sequential worker must notify all workers when it backtrack across a parallel phase and let each worker undo the conditional bindings it created during that parallel phase.

Heap-allocated vectors

Some variables, that would be allocated on the stack in a sequential WAM, are allocated in vectors stored on the heap. Consider, for example, the 'naive reverse' program:

```
nrev([],[]).
nrev([A|X],Y) :- nrev(X,Z), append(Z,[A],Y).
```

Here the variable Y on each recursion level is bound to the variable Z on the next recursion level. In a sequential WAM both Y and Z would be stored on the stack. This is not possible in our machine for two reasons. First, stacks are not shared between workers. Second, all instances of the variables are created *before* the parallel execution of the append calls is initiated, as a consequence of our execution model.

Hence all instances of the variables Y and Z are allocated in vectors on the heap rather than in binding environments on the stack. This is an example on the trade-off between efficient memory usage (stack allocation) and increased potential for parallelism (heap allocation). For tail-recursive programs this effect is even more dramatic.

The Reform engine also exploits heap-allocated vectors for another purpose. Consider the list in the first argument position of nrev/2 above. The sequential engine traverses the list and builds a vector of its elements before the parallel phase is initiated. This is necessary since we (*i*) need to know the final recursion depth (i.e. the length of the list) *before* the parallel phase is initiated, in order to determine how many parallel processes (recursion levels) to spawn, and (*ii*) need to index into the list during the parallel phase.

A problem is that after the parallel phase, the vector should be viewed as a linked list again. Our solution to this problem is to build a 'vector-list' by allocating the cons cells densely one after the the other. In this way we

can index into the list during the parallel phase, and still view it as a linked structure in the following sequential phase.

7.4 DATA SHARING

There are two aspects of data sharing between workers. First, terms created by the sequential worker (variables, numbers, structures and lists) must be accessible to parallel workers. Second, terms created by parallel workers must likewise be accessible to other parallel workers, and to the sequential worker.

Binding variables

The memory layout described above gives each worker the ability to refer to objects shared with other workers. This ability might lead to a situation where two workers try to simultaneously bind the same variable, potentially with the result of one binding destroying the other.

We therefore use an atomic exchange operation when binding a variable. If another worker has managed to bind the variable ahead of this worker, then the exchange operation will return that binding. The other workers binding must then be unified with this workers, to ensure consistency. A similar method is used in the implementation of Parallel NU- Prolog [103].

We have found that in our system this method is significantly faster than spin locking [93].

Creating shared structures

When building a structure on the heap other workers should not have access to the structure until it has been fully initialized. In WAM a structure is built using either put instructions or get instructions. When put instructions are used there is no problem, since no variable is bound to the structure until it has been fully initialized. A get instruction, on the other hand, might bind a variable to an incomplete structure and then proceed to fill in the missing parts using unify instructions.

We avoid references to uninitialized structures by modifying the get instructions so that they do not bind the variable, but save it for later binding. A new instruction has to be introduced after the last unify instruction (when the structure is complete) to bind the variable. This method is also discussed in Naish's paper on Parallel NU-Prolog [103].

Trailing

Variables must be trailed in the parallel phase, even though only safe programs are parallelized. There are two reasons for this.

1. There might be local nondeterminism within each recursion level. If that is the case, the worker must be able to backtrack locally. This forces the worker to, at least, trail bindings of local heap and stack variables ('local' variables reside in a data area managed by the worker).

2. A parallel worker might bind variables created before the parallel execution started. Since sequential backtracking is allowed across parallel calls, the machine must be able to undo bindings created during parallel phases.

The problem with trailing is that each worker has its own heap to work on. It is no longer possible to simply compare a variable's position on the heap with a heap pointer to determine whether it should be trailed or not. The situation gets even worse if we consider what might happen when the sequential worker continues executing after a parallel execution. In this case there might be variables distributed over all workers' heaps. A simple comparison between pointers cannot be used to determine whether a variable should be trailed or not, no matter how the heaps are arranged.

Our solution to this problem is to extend heap variables with a *timestamp*. This implies that the WAM has to be extended with a counter that is incremented whenever a choice point is created. This timestamp has to be saved in the choice point and restored on backtracking. A timestamp comparison is then used for determining whether to trail a variable or not.

7.5 PROCESS MANAGEMENT

Process management and scheduling are critical points in many of the parallel Prolog systems existing today.

In recursion-parallel systems much of the scheduling is done at compile time. It is then possible to determine which code is going to be executed in parallel. The number of processes executing the parallel code is determined immediately before parallel execution begins. These properties greatly simplify the process management problem; in fact, the scheduling overhead becomes negligible.

Suspension

The set of programs that can be parallelized in Reform Prolog has been restricted to those which are binding-deterministic with respect to shared variables. This condition is verified at compile time.

Recursion levels suspend only to preserve sequential semantics and safeness. To ensure sequential semantics, a recursion level suspends before binding

variables subject to time-sensitive tests; the level resumes when the variable is instantiated or it becomes leftmost. In both cases, preceding recursion levels cannot be affected by the binding.

To ensure safeness, a recursion level must not conditionally bind shared variables. In this situation, the level suspends until the variable is instantiated. If the level becomes leftmost, instantiation would occur and a safeness violation is signalled.

Some operations, notably general unification, unpredictably instantiate terms. Recursion levels suspend until leftmost before general unifications with terms containing time-sensitive variables (in the sense of above); if a general unification might create conditional bindings to shared variables, parallelization currently fails.

We implement these tests by busy-waiting. For instance, a worker suspending to preserve sequential semantics repeatedly checks if the variable has become instantiated or if the goal has become leftmost.

This method has the drawback that a suspended worker can tie up a processor for a long time. Whether this is a problem or not depends on the application program. If the average waiting time is short, then the overhead of this method is negligible.

The advantage of this method is that it does not slow down processes that do not wait for data—only the waiting worker is slowed down. No extra overhead is imposed on the Reform engine as compared to a sequential implementation.

Scheduling

One of the advantages of the Reform execution model is that much of the scheduling can be done at compile time. It is possible to determine at compile time which code is going to be executed in parallel, and the number of recursion levels to run can be determined prior to parallel execution. In the Reform engine this is done by examining the recursion argument.

The scheduling process is thus reduced to dividing recursion levels among workers. There are two different approaches to scheduling: static scheduling and dynamic scheduling.

Static scheduling. Static scheduling minimizes the need for synchronization. Most parallel Prolog systems cannot use static scheduling since too little information is available about the structure of the parallel execution before it is initiated. In recursion-parallel systems, information about which code is going to be executed, as well as the number of parallel processes,

is available. This makes it possible to distribute recursion levels to workers statically.

Of course, some programs are not well-suited for static scheduling, e.g., programs with poor load balancing due to significantly varying execution times of recursion levels.

Dynamic scheduling. The goal of dynamic scheduling methods is to optimize the trade-off between large process granularity and good load balance.

The simplest dynamic algorithm for scheduling is *self-scheduling* [138]. In this algorithm each processor executes one recursion level at a time until all levels have been executed. This method achieves almost perfect load balancing.

The problem with self-scheduling is, not surprisingly, that it tends to create too many processes with too fine granularity. However, this is less a problem in a recursion-parallel Prolog system than in loop-parallel Fortran systems, since the granularity of a single recursion level typically is greater than the granularity of a single iteration of a parallel loop.

More sophisticated dynamic algorithms has been proposed [80, 110, 92]. In these algorithms each processor is allocated a chunk of iterations at a time, instead of a single iteration. The chunk size may be fixed or variable. These algorithms have not yet been tested in Reform Prolog.

Task switching. Each worker is responsible for calculating which recursion levels it is going to execute. With static scheduling no synchronization is necessary to schedule work. With dynamic scheduling, on the other hand, it is necessary to synchronize accesses to a global variable holding the remaining number of unscheduled processes.

Task switching (i.e., starting a new process) amounts to a simple jump operation. With 'chunking' dynamic scheduling methods, the jump operation is preceded by an arithmetic calculation of how many processes to schedule in one chunk.

Mapping processes to processors. There are two simple ways to map recursion levels to processors regardless of whether we use static or dynamic scheduling (assuming chunks of more than one level in the latter case). Either consecutive recursion levels are mapped to consecutive processors (*horizontal* mapping), or consecutive recursion levels are mapped to the same processor (*vertical* mapping). If there are data dependencies

in the program, then horizontal mapping is to be preferred since that enables process pipelining. If there are no data dependencies, on the other hand, then it might be better to use vertical mapping since it improves data locality.

7.6 COMPILING RECURSION PARALLEL PROGRAMS

Compilation has two main components: ensuring safeness and introduce suspension and locking unification instructions where required. This is managed by combining three analyses: type inference, safeness analysis and locality analysis.

The type inference domain is an extension of the Debray-Warren domain [44], with the addition of support for lists and difference lists. The compiler uses both parallel and sequential types in code generation; parallel types hold at all times in the program, while sequential types hold when leftmost.

Safeness analysis investigates when the computation is in a nondeterminate, parallel state. Determinacy changes on procedure entry and cuts. Note that there is no attempt to prove determinacy of a goal; instead the analyzer merely records when determinacy may change. To get reasonable results, the analyzer simulates indexing on procedure entry.

Locality analysis has two tasks: to find terms that are local to one recursion level, and to mark shared terms subject to time-sensitive operations (such terms are called *fragile* since they must be handled with care). Local operations do not require suspension or locking unification; shared terms require locking unification but no suspension, while fragile terms may not be instantiated out of the sequential order. If the compiler were not to respect fragility, the system might stray from simulating sequential behavior [19].

The goal of locality analysis is to generate precisely WAM code for parallel operations on unshared data. When this is possible, there is no parallelization overhead once the parallel execution has started. That is, the compiler attempts to localize the overheads of parallel execution to the points where the full machinery is actually needed.

7.7 INSTRUCTION SET

The sequential WAM instruction set is extended with new instructions for supporting recursion-parallelism.

New instructions

The new instructions can be divided into five groups as follows.

Creating shared structures. These instructions create structures that might be accessed by other workers while they are created.

Creating vectors. These instructions build vector-lists that are used in the parallel phase. They are executed by the sequential worker. There are instructions that convert lists into vector-lists, as well as instructions that create vector-lists corresponding to binding environments.

Accessing global arguments. These instructions are used by the parallel workers to fetch data needed at their recursion levels from the sequential working space.

Process control. These instructions are used by the sequential worker to spawn parallel workers, and by parallel workers to switch from one recursion level to the next.

Runtime safeness tests. These instructions perform runtime tests to enforce safeness.

Examples

The following programs illustrate the use of some of the new instructions.

In the description of each instruction we use the term *step*, by which we understand the number of elements the recursive argument is reduced by in each recursive call. Some registers are denoted Gn, this refer to the sequential workers register Xn, which is globally accessible to all workers during a parallel phase.

Example 1:

```
        map([],[]).
        map([X|Xs],[Y|Ys]) :- foo(X,Y), map(Xs,Ys).
```

The program is compiled into the following extended WAM code.

```
map/2:  switch_on_term Lv La L1 fail

Lv:     try La
        trust L1

La:     % Sequential code for first clause of map/2
```

```
Lb:     % Sequential code for second clause of map/2

L1:     build_rec_poslist X0 X2 X3 X0   % first list
        build_poslist X1 X2 X4 X1       % second list
        start_left_body L2              % execute L2 in parallel
        execute map/2                   % call base case
L2:     initialize_left 1               % I := initial recursion level
L3:     spawn_left 1 X2 G2              % while (I++ < N) do X2 := I;
        put_nth_head G3 X2 0 X0         %  X0 := G3[I]
        put_nth_head G4 X2 0 X1         %  X1 := G4[I]
        call foo/2 0                    %  foo(G3[I],G4[I])
        jump L3                         % od
```

Let us describe the effects of the code for the recursive clause (label L1).

- `build_rec_poslist X0 X2 X3 X0` traverses the list in X0 and builds a vector-list of its elements, storing a pointer to it in X3, and storing the list length of X0 in X2. The last tail of the list is stored in X0.
- `build_poslist X1 X2 X4 X1` traverses the list X1 to its X2'th element, and builds a vector-list of its elements in X4. If the list has fewer than X2 elements and ends with an unbound variable, then it is filled out to length X2. The X2'th tail of the vector-list is unified with the X2'th tail of the list in X1, and finally X1 is set to the X2'th tail of the vector-list.
- `start_left_body L2` starts parallel execution of the foo/2 calls. The code at label L2 is run in parallel by all active workers. The sequential execution continues with the next instruction (execute map/2) when the parallel phase is finished.
- `initialize_left 1` initializes a worker for parallel execution. The step 1, given as argument, and the worker number is used for calculating the initial recursion level in static scheduling mode. In dynamic scheduling mode this instruction is ignored.
- `spawn_left 1 X2 G2` calculates the next recursion level for this worker and stores it in X2. The new level is calculated from the step 1 and the internal level count. If the new level is greater than the value stored in the global register G2 (i.e., the register X2 in the sequential worker), then then parallel computation is finished, otherwise the execution continues with the next instruction.
- `put_nth_head G3 X2 0 X0` performs the assignment X0 := G3[X2+0]. G3 points to a vector-list and X2+0 is the offset into the vector-list.
- `put_nth_head G4 X2 0 X1` similarly assigns X1 := G4[X2+0].

Example 2:

```
nrev([],[]).
nrev([X|Xs], Y) :- nrev(Xs,Z), append(Z,[X],Y).

append([],X,X).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

The program is compiled into the following extended WAM code.

```
nrev/2: switch_on_term fail La L1 fail  % fail cases never occur

   La:  get_nil X0
        get_nil X1
        proceed

   L1:  allocate
        build_rec_poslist X0 X3 X6 X0   % first list
        build_variables X1 X3 X5 X1     % second list
        /* code for saving X3, X5 and X6 in environment */
        call nrev/2, 0                  % call base case
        /* code for restoring X3, X5 and X6 from environment */
        start_right_body X3 L1          % execute L1 in parallel
        deallocate
        proceed                         % done
   L1:  initialize_right 1 G3           % I := initial recursion level
   L2:  spawn_right 1 X7                % while(I-- > 0) do X7 := I;
        allocate
        put_nth_head G5 X7 1 X0         %  X0 := G5[I+1]
        put_list X1                     %  X1 := [
        unify_nth_head G6 X7 0          %         G6[I]
        unify_nil                       %              ]
        put_nth_head G5 X7 0 X2         %  X2 := G5[I]
        call append/3, 0                %  append(G5[I+1],[G6[I]],G6[I])
        deallocate
        jump L2                         % od

append/3:
        await_nonvar X0                 % wait until first arg nonvar
        switch_on_term fail La L1 fail  % fail cases never occur

   La:  get_nil X0
        get_value X1 X2
        proceed

   L1:  get_list X0                     % X0 = [
        unify_variable X3               %       X|
        unify_variable X0               %         Xs]
```

```
lock_and_get_list X2 X4        % lock X2; X4 = [
unify_x_value X3               %                X|
unify_x_variable X5            %                  Zs]
unlock X2 X4                   % unlock X2; X2 = [X|Zs]
put_value X5 X2                % X2 := Zs
execute append/3               % append(X0,X1,X2)
```

We describe below the effects of the new instructions in the above code.

- `build_variables X1 X3 X5 X1` builds a vector-list containing X3+1 distinct unbound variables, storing a pointer to it in X5. A reference to the last variable in the vector-list is stored in X1.
- `start_right_body X3 L1` initiates parallel execution of the append/3 calls in the body of nrev/2. The code at label L1 is run in parallel by all workers. The sequential execution continues with the following instruction (deallocate) when the parallel phase is finished. The length of the recursion list is given in X3.
- `initialize_right 1 G3` initializes a worker for parallel execution. The step 1, given in the first argument, the length of the recursion list, given in G3, and the worker number is used for calculating the initial recursion level in static scheduling. In dynamic scheduling mode this instruction is ignored.
- `spawn_right 1 X7` calculates the next recursion level for this worker, using the step given in the first argument, and stores it in X7. If all recursion levels have been executed, the worker suspends and awaits the next parallel phase.
- `unify_nth_head G6 X7 0` writes the element G6[X7+0] onto the heap. G6 contains a pointer to a vector-list and X7+0 is the offset into the vector-list. This instruction never occurs in read mode.
- `await_nonvar X0` suspends until X0 contains a nonvariable or the recursion level has become leftmost in the resolvent.
- `lock_and_get_list X2 X4` checks the value in X2. If X2 contains a variable, it creates a list on the heap, stores a pointer to it in X4, and enters write mode. If X2 contains a list, the S register is set. Otherwise failure occurs.
- `unlock X2 X4` is ignored in read mode. In write mode, X2 and X4 are unified.

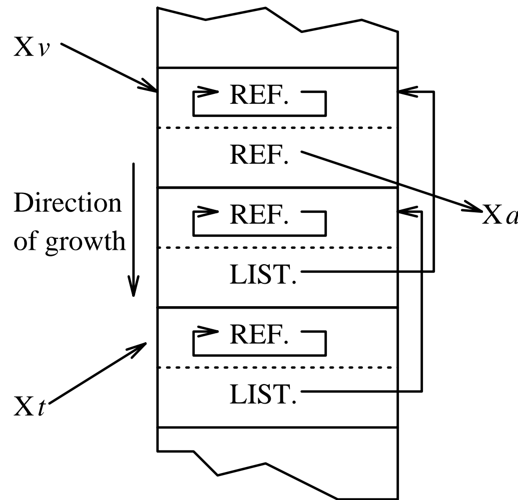


Figure 7.1: List (viewed as a vector) created by `build_neglist` .

Other Instructions. The extended instruction set contains some instructions not used in the two examples above. These instruction are described below.

We use the following notation. If x is a cons cell of a vector-list, then $x.tl$ denotes the tail of the cons cell.

- `build_poslist_value Xa Xn Xv Xt` This instruction is to `build_poslist` as `unify_ value` is to `unify_variable`.
- `build_neglist Xa Xn Xv Xt` A 'reverse list' vector-list of length Xn is created and stored in Xv. Xt is set to the head and Xa to the last tail of the vector-list, respectively. The last tail of the vector-list is in this case also the tail of the first element of the vector-list. See figure 7.1.
- `build_neglist_value Xa Xn Xv Xw Xt` is to `build_neglist` as `unify_value` is to `unify_variable`.
- `put_global_arg Gn Xi` stores the value of the sequential workers register Xn in Xi.
- `put_nth_tail Gv Xl O Xi` Similar to `put_nth_ head` but performs the assignment Xi := Gv[Xl+0].tl.
- `unify_nth_tail Gv Xl O` Similar to `unify_nth_head` but writes Gv[Xl+0].tl to the heap.

- `unify_global_arg Gg` writes the value of the sequential workers register Xg on the heap. This instruction never occurs in read mode.
- `await_leftmost` forces the current recursion level to suspend until it is leftmost in the resolvent, i.e., until all preceding recursion levels have terminated.
- `await_strictly_nonvar Xi` suspends the current recursion level until Xi contains a non-variable. If the recursion level becomes leftmost in the resolvent and Xi still contains an unbound variable, then a run-time error is signaled and execution fails.
- `await_variable Xi` suspends the current recursion level until it is leftmost in the resolvent. If the variable Xi becomes bound during suspension then a run-time error is signaled and the execution fails.
- `lock_and_get_structure F Xi Xn` Similar to `lock_and_get_list` but for structures with functor F.

7.8 EXPERIMENTAL RESULTS

In this section we present the results obtained when running some benchmark programs in Reform Prolog on a parallel machine.

Experimental methodology

Reform Prolog has been implemented on the Sequent Symmetry multiprocessor. This is a bus-based, cache-coherent shared-memory machine using Intel 80386 processors. The experiments described here were conducted on a machine with 26 processors, where we used 24 processors (leaving two processors for operating systems activities).

The metric we use for evaluating parallelization is the speedup it yields. We present *relative* and *normalized* speedups.

Relative speedup expresses the ratio of execution time of the program (compiled with parallelization) on a single processor to the time using p processors.

Normalized speedup expresses the ratio of execution time of the original sequential program (compiled without parallelization) on a single processor to the time using p processors on the program compiled with parallelization.

Benchmarks.

Programs and input data. We have studied the performance of four benchmark programs. One of the programs exploits independent AND-

parallelism and the others dependent AND-parallelism. Two programs are considerably larger than the others.

Map. This program applies a function to each element of a list producing a new list. In the measured program, the function simply decrements a counter 100 times. A list of 10,000 elements was used.

Nrev. This is the classic 'naive reverse' program run on a list of 900 elements.

Match. A dynamic programming algorithm for comparing, e.g., DNA-sequences. A sequence of length 32 was compared with 24 other sequences and the resulting similarity-values collected in a sorted binary tree.

Tsp. This program implements an approximation algorithm for the Traveling Salesman Problem. A tour of 45 cities was computed.

Load balance. One way of estimating the load balance of a computation is to measure the finishing time of the workers. We measured the execution time for each worker when executing our benchmarks. Static scheduling was used in all experiments.

Map. This program displayed a very uniform load balance (less than 0.3% difference between workers). This is hardly surprising since the number of recursion levels executed by each worker is large, and there is no difference in execution time between recursion levels.

Nrev. The execution time of each worker only varied about 3% when executing this program. There is a slight difference in the execution time of each recursion level but the large number of recursion levels executed by each worker evens out the differences.

Match. When 16 workers were used, 8 workers executed 2 recursion levels each, while 8 workers executed a single recursion level. This explains the relatively poor speedup on 16 workers. When 24 workers were used the execution time varied less than 0.3% between workers. This is explained by the fact that each worker executed one recursion level, and that all recursion levels executed in the same time.

Tsp. This program displayed an uniform load balance on all but three workers. This is explained by the fact that 45 recursion levels were executed in all; 21 workers executed 2 recursion levels each while 3 workers executed 1 recursion level each. Despite this the program displays good speedup (21.85). Using dynamic scheduling would not have improved the results in this case.

Sequential fraction of runtime. Parallelization occurs on a single level in Reform Prolog, and there are necessarily sequential startup portions of

the programs. The startup portions of our benchmark programs include setting up the arguments for the parallel call, large head unifications, and spawning parallel processes.

According to Amdahl's law, the time spent in the sequential part of the program will ultimately limit the speedup from parallelization.

The following table shows for each benchmark program how large fraction of the execution time on a sequential machine is not subject to parallelization.

Map	Nrev	String	Tsp
0.3%	0.04%	0.003%	0.005%

The unparallelized parts of the programs are negligible. As a consequence, we do not currently see a need for parallelizing head unifications of parallel predicates.

Results

The results of the experiments are summarized in the tables below. In the tables P stands for number of workers, T for runtime (in seconds), S_R for relative speedup, and S_N for normalized speedup. The sequential runtime for each program is given below each table.

P	T	S_R	S_N	T	S_R	S_N
1	40.40	1.00	0.98	30.80	1.00	0.88
4	10.12	3.99	3.89	8.08	3.81	3.43
8	5.07	7.96	7.76	3.96	7.77	6.99
16	2.54	15.91	15.50	2.01	15.32	13.78
24	1.70	23.76	23.15	1.36	22.65	20.36

Map. (39.59 sec.) **Nrev**. (27.70 sec.)

P	T	S_R	S_N	T	S_R	S_N
1	68.88	1.00	0.95	258.22	1.00	0.90
4	17.22	3.99	3.80	68.85	3.75	3.37
8	8.61	7.99	7.60	34.55	7.47	6.73
16	5.76	11.95	11.35	17.25	14.96	13.47
24	2.91	23.70	22.52	11.82	21.85	19.67

Match. (65.44 sec.) **Tsp**. (232.40 sec.)

Discussion

We briefly compare our system with Andorra-I, another compiler-based implementation supporting deterministic dependent AND-parallelism [118].

Note, however, that Andorra-I parallelizes a wider class of computations than does Reform Prolog. In particular, Andorra-I also supports OR-parallelism. The results reported for Andorra-I were obtained on a 10 processor Sequent Symmetry.

As before, take the parallel efficiency of a program to be the speedup on N processors divided by N. We have computed the efficiency relative to parallel Andorra-I, sequential Andorra-I and SICStus 2.1.

- Relative speedups on a set of 12 benchmarks range from 3.32 to 9.66, with a median of approximately 6.5 and a mean of approximately 6.4. The relative efficiency is taken to be 64%.
- Normalized speedups w.r.t. sequential Andorra-I on the same 12 benchmarks range from 2.04 to 7.26, with a median of 4.1 and a mean of 4.6. Normalized efficiency is then 46%.
- Compared with SICStus 2.1, Andorra-I was apparently 344 times faster in running the constraint solving fly_pan benchmark. Apart from this data point, on a range of 9 benchmarks, Andorra-I exhibited normalized speedups of 0.76 to 3.80 . The median was 1.47 and the mean 1.98. Normalized efficiency is thus 20%.

Only one benchmark can be directly compared to Reform Prolog, the nrv400 benchmark of naive reverse on 400 elements. The reported relative speedup is 8.25, the normalized speedup w.r.t. sequential Andorra-I is 6.55 and the normalized speedup w.r.t. SICStus 2.1 is 3.80. The efficiency is then 82%, 65% and 38%, respectively, on 10 processors. Reform Prolog has a relative efficiency of 95% and a normalized efficiency of 83% on 24 processors when running naive reverse on 900 elements.

We conclude that while Andorra-I can parallelize quite a wider range of programs, exploiting recursion-parallelism where available seems to have considerable benefits.

7.9 CONCLUSIONS

Reform Prolog implements parallelism across recursion levels by Reform compilation. One restriction is introduced on the recursive predicates subject for parallelization: bindings of variables shared between recursive calls of the predicate must be unconditional. This is not a severe restriction in practice.

The execution model has two major advantages:

First, a static process structure can be employed. That is, all parallel processes are created when the parallel computation is initiated. In most other

systems for parallel logic programming, processes can be dynamically created, rescheduled and destroyed during the parallel computation.

A consequence of the static process structure is that process management and scheduling can be implemented very efficiently. This opens up for high parallel efficiency (91–99% on the programs tested). Another consequence is that it is easy for the programmer to see which parts of the program are going to execute in parallel. This facilitates the task of writing efficient parallel programs.

Second, it is possible by global dataflow analysis to optimize the code executed by each parallel worker very close to ordinary sequential WAM code. This results in very low overheads for parallelization (2–12 % on the programs tested).

The apparent drawback of this approach is that not all available parallelism in programs are exploited. This is, however, a deliberate design decision: exploiting as much parallelism as possible is likely to lead to poor machine utilization on conventional multiprocessors.

Acknowledgments We thank the Swedish Institute of Computer Science (SICS) for making their 26 processor Sequent Symmetry available to us, and the Andorra-I team of Bristol University for providing us with their latest benchmarks results.

Compiler Optimizations in Reform Prolog: Experiments on the KSR-1 Multiprocessor

Thomas Lindgren, Johan Bevemyr and Håkan Millroth

in *Proc. EURO-PAR'95*, LNCS 966, Springer Verlag, 1995.

We describe the compiler analyses of Reform Prolog and evaluate their effectiveness in eliminating suspension and locking on a range of benchmarks. The results of the analysis may also be used to extract non-strict independent and-parallelism.

We find that 90% of the predicate arguments are ground or local, and that 95% of the predicate arguments do not require suspension code. Hence, very few suspension operations need to be generated to maintain sequential semantics. The compiler can also remove unnecessary locking of local data by locking only updates to shared data; however, even though locking writes are reduced to 52% of the unoptimized number for our benchmark set, this has little effect on execution times. We find that the ineffectiveness of locking elimination is due to the relative rarity of locking writes, and the execution model of Reform Prolog, which results in few invalidations of shared cache lines when such writes occur.

The benchmarks are evaluated on a cache-coherent KSR-1 multiprocessor with physically distributed memory, using up to 48 processors. Speedups scale from previous results on smaller, bus-based multiprocessors, and previous low parallelization overheads are retained.

8.1 INTRODUCTION

An important challenge in parallel processing is to keep parallelization overhead low: it does not make sense to use up two or more processors just to break even with sequential code. Another challenge is to make sure that designs scale with the size of the machine. The design of the Reform Prolog system is an attempt to meet these challenges without making programming much harder than in the sequential case.

Reform Prolog has been implemented on a varity of shared address space multiprocessors. The parallel implementation is designed as extension of a sequential Prolog machine [17]. The implementation imposes very little overhead for process management, such as scheduling and load balancing.

We have previously described the compilation scheme [98, 100], execution model [19, 20], and parallel abstract machine [19, 20] of Reform Prolog. In this paper we describe a set of compiler analyses and optimizations for increasing available parallelism and reducing parallelization overheads. We discuss the effectiveness of the optimizations and the runtime space/time efficiency of the compiler. The system is evaluated on a cache-coherent large-scale multiprocessor with physically distributed memory.

8.2 REFORM PROLOG

The Reform Prolog system supports a parallel programming model where a single conceptual thread of control is mapped to multiple low-level threads. Each thread runs an instance of the same recursive program in an asynchronous parallel computation. This model is often called SPMD (Single Program Multiple Data). The programming model is realized by a compilation technique that translates a regular form of recursion to a parallelizable form of iteration [98, 100].

Example. The following program compares a sequence B with a list of sequences. Each comparison, carried out by match/3, computes a similarity value V that is stored in a sorted tree T for later access. The tree is implemented as an incomplete data structure.

```
:- parallel match_seqs/3.
match_seqs([],_,_).
match_seqs([A|X],B,T) :-
    match(A,B,V),
    put_in_tree(T,V),
    match_seqs(X,B,T).
```

Assume that we invoke this program with a call containing an input list of four sequences. The programmer can then think of the recursive clause as being unfolded four times:

```
match_seqs([A1,A2,A3,A4|X],B,T) :-
    match(A1, B, V1), put_in_tree(T, V1),
    match(A2, B, V2), put_in_tree(T, V2),
    match(A3, B, V3), put_in_tree(T, V3),
    match(A4, B, V4), put_in_tree(T, V4),
    match_seqs(X,B,T).
```

Of course, this is not how Reform Prolog actually compiles the clause. However, the compiled code behaves *as if* the recursion is completely unfolded. When computing the call, four parallel processes are simultaneously spawned (the two calls within each process are executed sequentially):

$$\begin{array}{l} \text{match}(A_1, B, V_1), \text{put_in_tree}(T, V_1), \\ \text{match}(A_2, B, V_2), \text{put_in_tree}(T, V_2), \\ \text{match}(A_3, B, V_3), \text{put_in_tree}(T, V_3), \\ \text{match}(A_4, B, V_4), \text{put_in_tree}(T, V_4), \end{array}$$

The four calls to put_in_tree/2 are sequenced by synchronization on the shared variable T. However, the processes descend through the tree in parallel, temporally suspending when encountering not-yet-created subtrees. □

The parallel execution model of Reform Prolog restricts the nondeterministic behaviour of parallel programs so that the following properties hold [19, 20]:

- *Parallel programs obey the sequential semantics of Prolog.* This implies that time-dependent operations (type tests, etc.) on shared, unbound variables are carried out only when leftmost is the sequential computation order.
- *Parallel programs do not conditionally bind shared variables.* This is similar to binding determinism as defined by Naish [103] in that shared variables can only be bound when the process is deterministic. However, in contrast to Naish's binding determinism, nondeterministic bindings to *local* variables are allowed.

In order to ensure these properties, the Reform Prolog compiler performs a global dataflow analysis and generates code that suspends processes and

perform atomic updates only when necessary [85]. In particular, the compiler can generate precisely the code for a sequential Prolog machine when data are local.

8.3 COMPILER ANALYSES

The compiler analyses in the Reform Prolog compiler are based on *abstract interpretation* [40]. The abstract interpreter for Reform Prolog shares most of the characteristics of an abstract interpreter for sequential Prolog. This is natural, since each parallel process executes almost as a sequential Prolog machine.

We have modified Debray's dataflow algorithms [43, 45] for analysis of parallel recursive predicates. These algorithms compute call and success patterns for each procedure in the program. Call and success patterns describe the abstract values of the variables in a procedure call at procedure entry and exit, respectively.

The compiler carries out four different analyses using the same basic algorithm. The abstract domains of these analyses are described below.

Types

The type domain is similar to that of Debray and Warren [46], extended to handle difference lists [85]. For our present concerns it suffices to note that the type analysis can discover ground and nonvariable terms. The analysis precision is similar to that of Aquarius Prolog [155].

Aliasing and linearity

The analyzer derives possible and certain aliases by maintaining equivalence classes of possibly or certainly aliased variables. This is similar to the techniques used by Chang [28].

A term is *linear* if no variable occurs more than once in it. To improve aliasing information, the analyzer tracks whether terms are linear [73]. Three classes of linearity are distinguished: **linear**, **nonlinear**, and **indlist**. The latter denotes lists where elements do not share variables. The domain is thus:

$$\textbf{linear} \sqsubseteq \textbf{indlist} \sqsubseteq \textbf{nonlinear}$$

Consider a variable X that is 'split' into several subterms $\{X_1, \ldots, X_n\}$ by unification $X = [X_1, \ldots, X_n]$ or some similar operation. Assuming that there are no other live aliases of X in the rest of the clause, the analyzer uses linearity information as follows in this situation:

- If X is **linear**, then $X_1, \ldots, X_n$ are also **linear** and unaliased.
- If X is **indlist**, then $X_1, \ldots, X_n$ are **nonlinear** and unaliased.
- If X is **nonlinear**, then $X_1, \ldots, X_n$ are **nonlinear** and aliased.

Our domain does not express covering properties, so the compiler cannot exploit linearity information when there are still aliases of X alive in the current clause.

Linearity information is maintained for each equivalence class of aliases.

Determinism

The analyzer determines where each process may create a choice point. The compiler needs this information since it must keep track of variables that may be bound within the scope of a choice point. The determinism domain is quite simple:

$$\mathbf{det} \sqsubseteq \mathbf{nondet}$$

As long as there is no possibility that a process may have created a choicepoint, it has status **det**. When a choicepoint may have been created, the status is changed to **nondet**. Cuts reset the determinism status to a value saved when the predicate is entered, similarly to what is done in the concrete implementation.

To improve precision, the compiler uses *abstract indexing* to approximate the first-argument indexing that will occur at runtime. This technique selects the possible paths for the inferred types of the first argument, based on standard WAM indexing [159].

Locality

The analyzer maintains a hierarchy of data locality information:

- shared variables exposed to time-dependent operations by another process (clause indexing, arithmetic, type tests, etc.) are **fragile** and cannot be modified out of the sequential order;
- shared variables not subjected to time-dependent operations are **robust**;
- robust variables become **wbf** (will-be-fragile) when subjected to time-dependent operations—to subsequent processes, **wbf** variables will be seen as **fragile**;
- unshared data are **local**.

The locality domain is thus:

$$\textbf{local} \sqsubseteq \textbf{robust} \sqsubseteq \textbf{wbf} \sqsubseteq \textbf{fragile}$$

This locality domain can furthermore be used to detect non-strict independent and-parallelism [69]. As long as a process does not contain fragile data, it is independent of the results of other processes. Such independent programs can still construct shared data structures in parallel.

8.4 COMPILER OPTIMIZATIONS

The Reform Prolog compiler uses the information obtained by the analyzer for two purposes: verifying parallelizability and reducing parallelization overheads.

Our experience is that the compiler is able to filter out almost all unintended violations of the parallelizability conditions. Moreover, the analyzer can often help the programmer to identify code sections that cause unintended process suspensions.

Suspension and locking overheads can be reduced by optimizations that exploit the results of the compiler analyses.

Cost of process suspension

Suspension overheads occur in two situations, based on the execution model: (a) to preserve the sequential semantics, fragile variables cannot be bound unless the process is leftmost and (b) to ensure binding determinism, shared variables must not be bound conditionally.

The compiler can eliminate suspension instructions when data is **local** or known to be instantiated, or the process is deterministic and data are non-**fragile**.

Cost of locking unifications

Locking overheads occur when trying to bind possibly shared variables. Write accesses to shared variables must be locked to avoid data races and premature access to shared structures.

A locking write is required only when the heap cell being assigned is not **local**. In the WAM, writes (get/unify-instructions) are conditional depending on whether the accessed cell is bound or not. The compiler exploits type and locality information to strength-reduce locking instructions into non-locking ones where possible.

On the KSR-1, locking unification is done by simulating an atomic exchange operation. At present, this is done by locking the cache line of the variable to be bound, writing the variable and releasing the lock. A shared variable binding is then done as follows.

```
swap_x = Atomic_Exchange(x,y);
if (swap_x != x) {
    if (!unify(swap_x,y))
      Global_Failure();
}
```

Structures that may be bound to shared variables are constructed privately, and then atomically bound to the variable as described above.

If both suspension and locking overheads can be optimized away, then procedures called from a parallel predicate execute standard WAM code at full sequential speed.

8.5 EXPERIMENTAL SETUP

We have evaluated the system using three small (0.5 KB and 20-50 lines of code) and three medium-sized (3-12 KB and 100-425 lines of code) benchmark programs. The small benchmarks are:

- **map** A function is mapped over a list of 5000 elements, producing a new list. The function simply decrements a counter 1000 times.
- **nrev** A list of 1300 elements is reversed using the 'naive reverse' program.
- **tree** 50,000 elements are inserted into a sorted, incomplete, binary tree.

The medium-sized benchmarks are:

- **tsp** The travelling salesman problem of 48 cities is solved by an approximation algorithm that visits the nodes of the minimal spanning tree.
- **ga** A population of 48 individuals is evolved for 5 generations using a genetic algorithm. The application is the travelling salesman problem with 120 cities.
- **sm** A string is compared to 96 other strings using the Smith-Waterman string matching algorithm (a standard dynamic programming algorithm often used for comparing, e.g., DNA sequences). Each string contains 32 characters. Each comparison results in a similarity value, which is stored in a sorted, incomplete, binary tree.

The source code of the benchmark programs is available by ftp from ftp.csd.uu.se in the directory pub/reform/benchmarks.

The runtime statistics of the compiler were obtained on a Sun 630/MP with 4 40-MHz Sparc-2 processors and 64 MB of memory during a normal workday. The compiler was run in native code compiled SICStus Prolog version 2.1.6.

The performance measurements of the compiled parallel programs were obtained on a Kendell Square Research KSR-1 with 64 processors, each with 32 MB of memory. However, we were not able to allocate more than 48 processors due to external constraints.

8.6 COMPILER PERFORMANCE

The time required for analysis and total compilation time were (ms):

	Analysis	Total	Ratio
map	260	620	0.37
nrev	480	840	0.57
tree	850	1360	0.62
tsp	6149	8519	0.72
ga	7670	13870	0.55
sw	3850	5390	0.71

Thus, the analysis requires around 60-70% of the total compilation time; we consider this a reasonable overhead. Furthermore, the absolute compilation times (0.6 to 14 seconds) are quite reasonable, in particular when considering that the SUN 630/MP is not a particulary fast machine by today's standards. Aquarius Prolog seems to have similar absolute analysis times on similar hardware [57].

8.7 ANALYSIS RESULTS

We measured analysis results for arguments in procedures called from parallel predicates.

The following table shows the percentages of ground arguments and the locality information of non-ground arguments. The 'total' percentage is weighted with respect to the total number of predicate arguments in all benchmarks.

	ground	local	robust	wbf	fragile
map	60	40	-	-	-
nrev	25	-	12	12	51
tree	–	60	-	-	40
tsp	38	52	10	-	-
ga	35	59	2	-	4
sm	34	60	-	-	6
total	35	55	4	1	5

All benchmark programs are deterministic in the sense that they do not leave choicepoints when finished. Shallow backtracking does occur. The analyzer was able to verify these facts.

From these results we can see that:

- 90% of the arguments do not refer to data that require locking (i.e. the data are ground or local);
- 95% of the arguments do not refer to data that require suspension (i.e. the data are not fragile).

The 90% of arguments that do not require locking translates to a reduction to 52% of the original number of executed locking instructions.

8.8 PERFORMANCE OF COMPILED PROGRAMS

We measured execution times of sequential code and parallel code using up to 48 processors. The results (in seconds walltime) are:

	seq	1	2	6	12	24	48
map	81.6	81.8	40.9	13.8	6.98	3.60	1.86
nrev	18.7	22.6	12.5	4.11	2.20	1.13	0.82
tree	44.9	50.1	25.7	9.42	6.10	5.20	10.1
tsp	104	107	54.8	17.8	9.06	4.74	2.64
ga	77.0	82.0	40.9	13.9	7.68	4.24	3.40
sm	73.1	75.5	37.8	12.6	6.30	3.46	1.76

The speedups w.r.t. parallel code on one processor are:

	1	2	6	12	24	48
map	1	2.00	5.95	11.7	22.7	44.0
nrev	1	1.81	5.52	10.5	19.9	27.6
tree	1	1.95	5.31	8.20	9.63	4.97
tsp	1	1.95	6.00	11.8	22.6	40.5
ga	1	2.00	5.90	10.7	19.3	24.1
sm	1	2.00	6.00	12.0	21.8	42.9

The speedups w.r.t. sequential code are:

	1	2	6	12	24	48
map	1.00	1.99	5.93	11.7	22.7	43.9
nrev	0.83	1.50	4.55	8.66	16.6	22.8
tree	0.90	1.75	4.77	7.37	8.60	4.50
tsp	0.98	1.91	5.87	11.6	22.1	39.6
ga	0.94	1.88	5.53	10.0	18.2	22.6
sm	0.97	1.93	5.81	11.6	21.1	41.5

We see that the parallelization overhead is very low: 0–17%, with the larger benchmarks in the range of 2–6%.

The absolute speedup is very good on three benchmarks (map,tsp,sm), ordinary on two programs (nrev,ga) and bad on one program (tree). The three programs with ordinary or bad speedup all suffer from lack of exploitable parallelism: the nrev and tree programs are sequentialized by fragile variables, and the ga program invokes a sequential sort routine after the parallel computation (this limits the speedup according to Amdahl's law).

The key characteristics of the programs are:

map	no suspension and no communication
nrev	heavy suspension and communication
tree	almost only suspension and communication
tsp	some communication and little suspension
ga	sequential sort after parallel computation
sm	some suspension and little communication

8.9 EFFECTIVENESS OF OPTIMIZATIONS

To measure the effectiveness of optimizations (elimination of suspension and locking instructions) we compiled the benchmark programs without exploiting global analysis.

Elimination of suspension instructions

To evaluate the effectiveness of removing suspension instructions, we rewrote the benchmarks to suspend whenever unbound heap cells were read by time-dependent operations, or written. Since there is no knowledge on what data is shared or fragile, this is required to retain sequential semantics.

The execution times without analysis were the following. We were unable to allocate 48 processors for this experiment and so show execution times for up to 24 processors.

Program	1	2	6	12	24
map	90.9	45.3	15.4	7.90	4.10
nr	30.5	34.4	37.0	42.3	51.3
tree	49.3	30.0	9.90	5.80	5.80
tsp	123.4	126	136	148	197
ga	74.5	37.7	14.1	7.70	6.10
sm	86.9	87.8	93.0	103	131

On 24 processors, there is comparable efficiency in the map, tree and ga benchmarks. For the map benchmark, the parallel computation consists of independent and trivial work that does not allocate heap data. The tree program can only write data when leftmost and so is essentially the same with or without analysis. The ga program performs all computations using scalar or ground data; all unbound variables are found on the stack. Since data on the stack cannot be shared, the engine can determine the outcome of suspension instructions even without analysis. The presence of extra suspension operations still has a cost: on 24 processors, the analyzed version of ga is 44% faster than the unanalyzed one.

Nrev, tsp and sm show net slowdowns when parallelized without global analysis. Tsp and sm perform considerable local work, which the no-analysis version delays until leftmost almost immediately and so sequentializes the program. For nrev without analysis, only the leftmost worker of nrev can write data at any time. With analysis, a 'pipeline' of processes appears and allows quicker completion.

Our conclusion is that there are substantial benefits to detecting computations local to a process (e.g., tsp, sm), and that more complex communication patterns (e.g., the nrev 'pipeline' effect) can be exploited transparently by the compiler.

Elimination of locking instructions

The elimination of locking instructions is ineffective: there is no measurable difference in execution times when the number of locking unifications are reduced to 52% of the original number. Two factors contribute to this phenomenon:

First, locking instructions are infrequent even in unoptimized code. Locking is spatially infrequent, since assignments to heap variables is a small fraction of the total amount of data written in Prolog implementations [144, 56].

Locking is temporally infrequent, since our Prolog implementation is based on byte-code emulation of WAM [159] instructions. In unoptimized code, 2100–3700 machine instructions were executed for each locking operation on the three larger benchmarks.

Second, single locking instructions are, on average, fast due to cache organization: cache lines owned by a single processor do not need global invalidations. Shared cache lines are infrequently invalidated by our programs since shared variables are written at most once (bindings of shared variables cannot be undone by backtracking). This means that a shared cache line is invalidated at most k times, where k is the number of variables per line ($k = 4$ in our implementation on the KSR-1).

8.10 CONCLUSION

We have described compiler analyses that allow us to:

- verify that 95% of the predicate arguments do not refer to data that require suspension instructions;
- verify that 90% of the predicate arguments do not refer to data that require locking instructions.

The elimination of suspension instructions is very effective, since it makes otherwise quite sequential programs highly parallel. However, the elimination of locking instructions is ineffective, in that there is no measurable difference in execution times when locking instructions are removed. The most important contributing factor to this effect is that the single-assigment, non-backtrackable shared variables of Reform Prolog result in few invalidations of shared cache lines.

This cache behaviour is notable, since an increasingly important problem in parallel processing is to hide the latencies of physically distributed and comparatively slow memories. It is reasonable to expect that most future single adress-space architectures will have distributed memories and to expect a continuing increase in processor to memory speed ratio [109].

The performance measurements on the KSR-1 can be summarized as follows.

- Low parallelization overhead (0–17%, with the larger benchmarks in the range of 2–6%).
- Good absolute parallel efficiency on 48 processors (82–91%) provided that there is enough parallelism in the program.

Our data indicate that each process executes in a mostly sequential fashion: suspension and locking is rare. Hence, sequential compiler technology should be largely applicable to our system. We intend to employ such techniques to improve absolute execution speeds further. Future work also includes a detailed quantitative characterization of the memory system behaviour of Reform Prolog programs on different architectures.

Acknowledgements. We thank the University of Manchester for providing access to their KSR-1 computer.

8.11 APPENDIX A: ANALYSIS MEASUREMENT DATA

Predicates and total number of predicate arguments in benchmark programs.

Program	Predicates	Total args
map	3	5
nrev	3	6
tree	2	5
tsp	15	79
ga	19	128
sm	9	35

Type domain. The abstract domain used by the analyzer is that of Debray and Warren:

$$\bot \sqsubseteq \mathbf{gnd} \sqsubseteq \mathbf{nv} \sqsubseteq \mathbf{any}$$

$$\bot \sqsubseteq \mathbf{free} \sqsubseteq \mathbf{any}$$

with the addition of lists on the form

list_{any,gnd,nv,free}_{free,nil,free+nil} nil, **free+nil**

Using this expanded domain allows the analyzer to derive precise types for programs using difference lists. The analyzer uses list types to derive better information for list-recursive loop bodies, and can use nonvariable terms to eliminate some repeated suspension instructions (e.g., due to indexing on the first argument, followed by unification once the clause is selected).

Information derived by analyzer. We measured only predicates that execute during the parallel phases of the programs, “parallel predicates”, since only they were affected by the optimizations described in the paper.

	gnd	nv	free	list_*_n	list_*_f	any
map	3	-	2	-	-	-
nrev	1	-	2	1	2	-
tree	-	-	-	-	-	5
tsp	27	2	18	5	2	25
ga	36	-	47	16	6	23
sm	10	7	5	3	-	10

Locality information derived by analyzer for parallel predicates.

	pred	args	gnd	local	robust	wbf	fragile
map	2	4	2	2	-	-	-
nrev	2	6	2	-	1	1	2
tree	2	5	-	3	-	-	2
tsp	15	79	31	40	8	-	-
ga	19	128	45	75	2	-	6
sm	9	35	12	21	-	-	2

Bibliography

1. H. Aït-Kaci, *The WAM: A (Real) Tutorial*, MIT Press, 1991. {5}

2. K.A.M. Ali, R. Karlsson, The Muse or-parallel Prolog model and its performance, in *Proceedings of the North American Conference on Logic Programming*, MIT Press, 1990. {3}

3. A.W. Appel, A runtime system, *Lisp and Symbolic Computation* 3(4), 1990. {74}

4. A.W. Appel, Simple generational garbage collection and fast allocation, *Software—Practice and Experience* 19(2):171–183, 1989. {10, 74, 78}

5. A.W. Appel, *Compiling With Continuations*, Cambridge University Press, 1992. {16, 25, 27, 37}

6. A.W. Appel, Z. Shao, Callee-saves registers in continuation-passing style, *Lisp and Symbolic Computation*, pp. 191-221, 1992. {37}

7. A.W. Appel, Z. Shao, An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures, Princeton University CS-TR-450-94, Princeton University, 1994. {37}

8. K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin, Garbage Collection for Prolog Based on WAM, *Communications of the ACM*, 31(6):719–741, June 1988. {19, 20, 72, 74, 80}

9. U. Banerjee, R. Eigenmann, A. Nicolau, D.A. Padua, Automatic program parallelization, Proceedings of the IEEE, vol. 81, no. 2, February 1993 {22}

10. J. Barklund, J. Bevemyr, Executing bounded quantifications on shared memory multiprocessors, in *Programming Language Implementation and Logic Programming 1993*, LNCS 714, Springer Verlag, 1993. {22}

11. J. Barklund, H. Millroth, *Nova Prolog*, UPMAIL Technical Report 52, Computing Science Department, Uppsala University, 1988. {4, 22}

12. J. Barklund, H. Millroth, Providing iteration and concurrency in logic programs through bounded quantifications, in *Proc. International Conference on Fifth Generation Systems*, Ohmsha, 1992. {4, 22}

13. J. Barklund, H. Millroth, Garbage cut for garbage collection of iterative Prolog programs, *3rd Symposium on Logic Programming*, Salt Lake City, September 1986, IEEE. {19, 73}

14. J. Beer, The occur-check problem revisited, Journal of Logic Programming Vol. 5, pp. 243-261, North-Holland, 1988. {17, 56, 68}

15. Y. Bekkers, O. Ridoux and L. Ungaro, Dynamic Memory Management for Sequential Logic Programming Languages, *Proceedings of the International Workshop on Memory Management 92*, LNCS 637, Springer-Verlag, Berlin, 1992. {19, 73}

16. G. Bell, Ultracomputers: a Teraflop before its time, in *Communications of the ACM*, Vol. 35, No. 8, 1992. {85}

17. J. Bevemyr, *A Recursion Parallel Prolog Engine*, Licentiate of Philosophy Thesis, Uppsala Theses in Computer Science 16/93, 1993. {13, 92, 122, 152, 163, 166, 167}

18. J. Bevemyr, A scheme for executing nested recursion parallelism, in JICSLP'96 Post-Conference Workshop on Implementation Techniques, September 1996. {15}

19. J. Bevemyr, T. Lindgren, H. Millroth, Exploiting recursion-parallelism in Prolog, in *PARLE-93*, eds. A. Bode, M. Reeve, G. Wolf, LNCS 694, Springer Verlag, 1993. {110, 122, 123}

20. J. Bevemyr, T. Lindgren, H. Millroth, Reform Prolog: The language and its implementation, in *Logic Programming: Proceedings of the Tenth International Conference*, MIT Press, 1993. {92, 122, 123}

21. J. Bevemyr, T. Lindgren, Simple and efficient copying garbage collection for Prolog, in Programming Language Implementation and Logic Programming 1994, LNCS 844, Springer Verlag, 1994. {37}

22. P. Bigot, D. Gudeman, S.K. Debray, Output value placement in moded logic programs, in *Logic Programming: Proceedings of the Eleventh International Conference*, MIT Press, 1994. {18, 56, 69}

23. K. De Bosschere, S.K. Debray, D. Gudeman, S. Kannan, Call forwarding: A simple interprocedural optimization technique for dynamically typed languages, in *Proc. Principles of Programming Languages*, ACM Press, 1994. {18, 40, 63, 69}

24. P. Brisset, O. Ridoux, Continuations in λProlog, in *Proceedings of the Tenth International Conference on Logic Programming*, ed. D.S. Warren, MIT Press, 1993. {16, 27}

25. M. Carlsson, On Implementing Prolog in Functional Programming, NGC 2 (1984), pp. 347-359. {17, 28}

26. M. Carlsson, *Design and Implementation of an Or-Parallel Prolog Engine*, Ph.D. Thesis, SICS-RITA/02, 1990. {2, 151}

27. M. Carlsson, On the efficiency of optimizing shallow backtracking in compiled Prolog, in *Proc. Sixth International Conference on Logic Programming*, MIT Press, 1989. {2}

28. J.-H. Chang, *High performance execution of Prolog programs based on a static dependency analysis*, Ph.D. Thesis, UCB/CSD 86/263, Univ. Calif. Berkeley, 1986. {21, 24, 124}

29. T. Chen, I.V. Ramakrishnan, R. Ramesh, Multistage indexing algorithms, in *Proc. Joint International Conference & Symposium on Logic Programming'92*, MIT Press, 1992. {2, 52}

30. C.J. Cheney, A nonrecursive list compacting algorithm, *Communications of the ACM*, 13(11):677–678, November 1970. {9, 10, 72, 75}

31. D.A. Clark, C.J. Rawlings, J. Shirazi, L-L. Li, K. Schuerman, M. Reeve, A. Véron, Solving large combinatorial problems in molecular biology using the ElipSys parallel constraint logic programming system, in Computer Journal 36(4). {3}

32. K.L. Clark, F. McCabe, The control facilities of IC-Prolog, in *Expert Systems in the Micro-Electronic World* (ed. D. Michie), Edinburgh University Press, 1979. {21}

33. K.L. Clark, S. Gregory, A relational language for parallel programming, in *Proceedings ACM Symposium on Functional Programming and Computer Architecture*, 1981. {21}

34. K.L. Clark, S. Gregory, PARLOG: A parallel logic programming language, report DOC 83/5, Department of Computing, Imperial College, London, 1983. {4, 21}

35. M. Codish, A. Mulkers, M. Bruynooghe, M. Garcia de la Banda, M. Hermenegildo, Improving abstract interpretations by combining domains, in *Proceedings of the 1993 Symposium on Partial Evaluation and Program Manipulation*, ACM Press, 1993. {24}

36. J. Cohen, Garbage Collection of Linked Data Structure, *Computing Surveys*, 13(3):341–367, September, 1981. {19, 72}

37. A. Colmerauer, H. Kanoui, R. Pasero, P. Roussel, Un Système de Communication Homme-Machine en Français, Groupe de Recherche en Intelligence Artificielle, Univ. de Aix-Marseille, Luminy, 1972. {1}

38. J.S. Conery, D.F. Kibler, Parallel interpretation of logic programs, in *Proceedings ACM Symposium on Functional Programming and Computer Architecture*, 1981. {4, 20}

39. A. Cortesi, G. Filé, W. Winsborough, Prop revisited: propositional formulas as abstract domain for groundness analysis, in *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science*, IEEE Press, 1991. {24}

40. P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, 1977. {124}

41. P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, Journal of Logic Programming, 1992:13:103-179. {18, 69}

42. S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, D.S. Warren, Unification factoring for efficient execution of logic programs, in *Proc. Principles of Programming Languages*, MIT Press, 1996. {2}

43. S.K. Debray, Static inference of modes and data dependencies in logic programs, ACM Transactions on Programming Languages and Systems, Vol. 11, No. 3, July 1989, pp. 418-450. {23, 124}

44. S.K.Debray, A simple code improvement scheme for Prolog, Journal of Logic Programming, 1992:13:57-88. {5, 16, 17, 28, 51, 91, 110}

45. S.K. Debray, Efficient dataflow analysis of logic programs, Journal of the ACM, Vol 39, No. 4, October 1992. {23, 124}

46. S.K. Debray, D.S. Warren, Automatic mode inference for logic programs, Journal of Logic Programming, Vol. 5, No. 3, 1988. {23, 124}

47. S.K. Debray, personal communication, 1995. {51}

48. S.K. Debray, T. Proebsting, Inter-procedural control flow analysis of first order programs with tail call optimization, Draft, University of Arizona, May 1996. {17}

49. D. De Groot, Restricted AND-parallelism, in *Proceedings of the International Conference on Fifth Generation Systems*, North-Holland, Amsterdam, 1984. {4, 20}

50. B. Demoen, A. Mariën, Implementation of Prolog as Binary Definite Programs, Proceedings of the Second Russian Conference on Logic Programming, Springer Verlag. {5, 16, 27}

51. B. Demoen, On the Transformation of a Prolog Program to a More Efficient Binary Program, Proceedings of the LOPSTR'92 workshop, Manchester, July 1992. {5, 16, 27}

52. B. Demoen, G. Engels, P. Tarau, Segment order preserving copying garbage collection for WAM based Prolog, in ACM Symposium on Applied Computing, ACM Press, 1995. {20}

53. A. Diwan, D. Tarditi, E. Moss, *Memory Subsystem Performance of Programs with Intensive Heap Allocation*, Technical report CMU-CS-93-227, Carnegie-Mellon University, 1993. {37}

54. P. Feautrier, Dataflow analysis of array and scalar references, International Journal of Parallel Programming, Vol. 20, No. 1, February 1991, Plenum Press {22}

55. M. Felleisen, *Transliterating Prolog into Scheme*, Computer Science Department Technical Report 182, Indiana University, 1985. {17, 28}

56. M.A. Friedman, *A characterization of Prolog execution*, Ph.D. Thesis, University of Wisconsin at Madison, 1992. {39, 131}

57. T. Getzinger, Abstract Interpretation for the Compile-Time Analysis of Logic Programs, Ph.D. Thesis, Technical Report ACAL-TR-93-09, University of South California, September 1993. {8, 18, 56, 57, 69, 128}

58. T. Getzinger, The Costs and Benefits of Abstract Interpretation-Driven Prolog Optimization, in *Proc. First International Static Analysis Symposium*, LNCS 864, Springer Verlag, 1994. {18, 23, 56, 69}

59. D. Gudeman, K. De Bosschere, S.K. Debray, `jc`: an efficient and portable sequential implementation of Janus, in *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, 1992. {8}

60. G. Gupta, V. Santos Costa, R. Yang, M.V. Hermenegildo, IDIOM: Intergrating Dependent and-, Independent and- and Or-parallelism, in *Logic Programming: Proceedings of the 1991 International Symposium*, MIT Press, 1991. {5, 21}

61. G. Gupta, B. Jayaraman, Compiled and-or parallel execution of logic programs, in *Proc. North American Conference on Logic Programming '89*, MIT Press, 1989. {5}

62. G. Gupta, B. Jayaraman, Optimizing and-or parallel implementations, in *Proc. North American Conference on Logic Programming '90*, MIT Press, 1990. {5}

63. G. Gupta, M.V. Hermenegildo, ACE: And/Or-parallel copying-based execution of logic programs, in *Parallel Execution of Logic Programs*, LNCS 569, Springer Verlag, 1991. {4, 5, 15, 21}

64. S. Haridi, A logic programming language based on the Andorra model, New Generation Computing 7(1990), pp. 109-125, Springer Verlag, 1990. {21}

65. W.L. Harrison III, The interprocedural analysis and parallelization of Scheme programs, Lisp and Symbolic Computation, Vol. 2, no. 3/4, 1989 {22}

66. C. Haynes, Logic continuations, Journal of Logic Programming, 4(2):157–176, June 1987. {17, 28}

67. M.V. Hermenegildo, An abstract machine for restricted AND-parallel execution of logic programs, in *Third International Conference on Logic Programming*, LNCS 225, Springer Verlag, 1986. {4, 20}

68. M.V. Hermenegildo, K.J. Greene, &-Prolog and its performance: exploiting independent and-parallelism, in *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press, 1990. {4, 15, 20}

69. M.V. Hermenegildo, F. Rossi, Non-strict independent and-parallelism, in *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press, 1990. {4, 20, 126}

70. M. Hermenegildo, M. Carro, Relating data-parallelism and (And-)parallelism in logic programs, Computer Languages Journal, to appear. {15}

71. D. Jacobs, A. Langen, Accurate and efficient approximation of variable aliasing in logic programs, in *Proc. North American Conference on Logic Programming 1989*, MIT Press, 1989. {24}

72. G. Janssens, M. Bruynooghe, Deriving descriptions of possible values of program variables by means of abstract interpretation, Journal of Logic Programming 1992:13:205-258. {24}

73. N. Jones & H. Søndergaard, A semantics-based framework for the abstract interpretation of Prolog, report 86/14, University of Copenhagen, 1986. {24, 124}

74. N. Jones, C. Gomard, P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993. {17}

75. K. Kahn, M. Carlsson, How To Implement Prolog on a Lisp Machine, in *Implementations of Prolog*, ed. J. Campbell, Ellis Horwood, 1984. {17, 28}

76. K. Kahn, M. Carlsson, *The Compilation of Prolog Programs without the Use of a Prolog Compiler*, UPMAIL Technical Report 27, Uppsala University, 1984. {17, 28}

77. E. Kohlbecker, *eu-Prolog*, Technical Report 155, Computer Science Department, Indiana University, 1984. {17, 28}

78. R.A. Kowalski, Predicate logic as a computer language, in *Information Processing 74*, pp. 569-574, North-Holland, 1974. {1}

79. D. Kranz, *Orbit: An Optimizing Compiler for Scheme*, Doctoral Thesis, Yale University, 1988. {25, 38}

80. C.P. Kruskal, A. Weiss, Allocating Independent Subtasks on Parallel Processors, IEEE Transactions on Software Engineering, Vol. 11, No. 10, 1985. {109}

81. J.R. Larus, P.N. Hilfinger, Detecting conflicts between structure accesses, in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, ACM Press, 1988 {23}

82. J.R. Larus, P.N. Hilfinger, Restructuring Lisp programs for concurrent execution, in *Proceedings of the ACM/SIGPLAN PPEALS 1988 Parallel Programming: Experience with Applications, Languages and Systems*, ACM Press, 1988 {23}

83. S. Le Huitouze, A new datastructure for implementing extensions to Prolog, Proc. Programming Language Implementation and Logic Programming 1990, LNCS 456, Springer Verlag, 1990. {19}

84. H. Lieberman, C. Hewitt, A real-time garbage collector based on the lifetimes of objects, Communications of the ACM, 26(6):419–429, June 1983. {10, 78}

85. T. Lindgren, *The compilation and execution of recursion-parallel Prolog on shared-memory multiprocessors*, Licentiate of Philosophy Thesis, Uppsala Theses in Computer Science 18/93, November 1993. {13, 92, 124, 150, 151, 152}

86. T. Lindgren, A continuation-passing style for Prolog, in *Proc. International Logic Programming Symposium 1994*, MIT Press, 1994. {40, 41}

87. T. Lindgren, Control flow analysis of Prolog (extended remix), Technical Report 112, Uppsala University, 1995. {42}

88. T. Lindgren, Compiling for nested recursion-parallelism, in JICSLP'96 post-conference workshop on implementation, September 1996. {15}

89. J. Lloyd, *Foundations of Logic Programming* (2nd ed.), Springer Verlag, {2}

90. E. Lusk, D.H.D. Warren, S. Haridi, P. Brand, R. Butler, A. Calderwood, M. Carlsson, A. Ciepielewski, T. Disz, B. Hausman, R. Olson, R. Overbeek, R. Stevens, P. Szeredi, The Aurora or-parallel system, New Generation Computing, vol 7(2-3), 1990. {3}

91. A. Mariën, B. Demoen, A new scheme for unification in WAM, in *International Logic Programming Symposium 1991*, MIT Press, 1991. {2}

92. E.P. Markatos & T.J. LeBlanc, Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors, Technical Report 410, University of Rochester, March 1992. {109}

93. J.M. Mellor-Crummey & M.L. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM Transactions on Computer Systems, Vol 9, Febr., 1991. {106}

94. M. Meier, Recursion vs. iteration in Prolog, in *Proc. Eighth Intl. Conf. on Logic Programming*, MIT Press, 1991. {52}

95. M. Meier, Compilation of compound terms in Prolog, technical report ECRC-95-12, ECRC, July 1990. {2}

96. M. Meier, Shallow backtracking in Prolog programs, technical report ECRC-95-11, ECRC, February 1987. {2}

97. H. Millroth, *Reforming the compilation of logic programs*, Ph.D. Thesis, Uppsala Theses in Computer Science 10, 1991. {11}

98. H. Millroth, Reforming compilation of logic programs, in *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press, 1991. {122}

99. H. Millroth, Reform compilation for non-linear recursion, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LNCS 624, Springer Verlag, 1992. {11}

100. H. Millroth, SLDR-resolution: parallelizing structural recursion in logic programs, Journal of Logic Programming, Vol 25(2), Nov. 1995, pp. 93-117. {4, 11, 86, 102, 122, 149}

101. F. Morris, A Time- and Space- Efficient Compaction Algorithm, Communications of the ACM, 12(9):662–665, August 1978. {19, 72}

102. K. Muthukumar, M.V. Hermenegildo, Compile-time derivation of variable dependency using abstract interpretation, Journal of Logic Programming, 1992:13:315-347. {21, 24}

103. L. Naish, Parallelizing NU-Prolog, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, 1988. {4, 13, 14, 21, 86, 103, 106, 123}

104. U. Neumerkel, A transformation based on the equality between terms, in *Logic Programming Synthesis and Transformation, LOPSTR'93*, Springer Verlag, 1993. {16}

105. U. Neumerkel, Continuation Prolog: A new intermediary language for WAM and BinWAM code generation, in Post-ILPS'95 Workshop on Implementation of Logic Programming Languages. {16}

106. U. Nilsson, Towards a methodology for the design of abstract machines for logic programming languages, Journal of Logic Programming, 1993:16:163-189. {16, 27}

107. W.J. Older and J.A. Rummell, An Incremental Garbage Collector for WAM-Based Prolog, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, 1992. {19, 73}

108. D. Palmer and L. Naish, NUA-Prolog: An Extension to the WAM for Parallel Andorra, *Proc. 8th Int. Conf. Logic Programming*, Paris, MIT Press, 1991. {96}

109. D.A. Patterson & J.L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann, 1993. {132}

110. C.D. Polychronopoulos, D.J. Kuck, Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, IEEE Transactions on Computers, Dec. 1987. {109}

111. E. Pontelli, G. Gupta, Data Parallel Logic Programming in &ACE, Proc. of the IEEE International Symposium on Parallel and Distributed Processing, IEEE Press, 1995. {15}

112. E. Pontelli, G. Gupta, Determinacy Driven Optimization of And-Parallel Prolog Implementations, *Int. Conf. Logic Programming*, MIT Press, 1995. {15}

113. G. Puebla, M. Hermenegildo, Implementation of multiple specialization in logic programs, in *Proc. Partial Evaluation and Program Manipulation*, ACM Press, 1995. {18, 69}

114. J. A. Robinson, A machine-oriented logic based on the resolution principle, Journal of the ACM, 12(1):23-41, 1965. {1}

115. D. Sahlin, Making garbage collection independent of the amount of garbage, Research Report R87008, Swedish Institute of Computer Science, 1987. {20, 74}

116. D. Sahlin, M. Carlsson, Variable shunting for the WAM, SICS research report R91:07, SICS, 1991. {19}

117. D. Sahlin, The Mixtus approach to automatic partial evaluation of full Prolog, in *Proc. North American Conference on Logic Programming'90*, MIT Press, 1990. {18, 69}

118. V. Santos Costa, D.H.D. Warren, R. Yang, Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism, in *Third ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, ACM Press, 1991. {5, 21, 96, 97, 118}

119. V. Santos Costa, D.H.D. Warren, R. Yang, The Andorra-I preprocessor: supporting full Prolog on the basic Andorra model, in *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press, 1991. {21}

120. V. Santos Costa, D.H.D. Warren, R. Yang, The Andorra-I engine: a parallel implementation of the basic Andorra model, in *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press, 1991. {21}

121. H. Schorr and W.M. Waite, An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, Communications of the ACM, 10(8):501–506, August 1967. {19, 72}

122. R. Sedgewick, *Algorithms (2nd edition)*, Addison-Wesley, 1989. {7}

123. D.C. Sehr, L.V. Kale, D.A. Padua, Loop transformations for Prolog programs, LNCS 768, Springer Verlag, 1993. {17, 22, 23, 51}

124. D.C. Sehr, *Automatic Parallelization of Prolog Programs*, Ph.D. Thesis, Univ. of Illinois at Urbana Champaign, 1992. {17, 22, 23, 51}

125. Z. Shao, A.W. Appel, *Space-Efficient Closure Representations*, Princeton University CS-TR-454-94, Princeton University. {37}

126. E.Y. Shapiro, *A subset of Concurrent Prolog and its interpreter*, ICOT Technical Report TR-003, Institute for New Generation Computing Technology, Tokyo, 1983. {4, 21}

127. K. Shen, Exploiting dependent and-parallelism in Prolog: the dynamic dependent and-parallel scheme (DDAS), in *Proceedings of the Joint International Symposium on Logic Programming*, MIT Press, 1992. {4, 12, 13, 21}

128. K. Shen, Initial results of the parallel implementation of DASWAM, in *Logic Programming: Proceedings from the 1996 Joint International Conference and Symposium*, MIT Press, 1996 {4, 158}

129. K. Shen, *Studies of And-Or Parallelism*, Ph.D. Thesis, Cambridge University, revised June 1992. {4, 12, 13, 21}

130. O. Shivers, Control flow analysis in Scheme, in *Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation*, ACM Press, 1988. {17, 35, 51}

131. J.P. Singh, J.L. Hennessy, An empirical investigation of the effectiveness and limitations of automatic parallelization, in *Proceedings of the International Symposium on Shared Memory Multiprocessing*, Tokyo, April 1991 {22}

132. D.A. Smith, MultiLog: data or-parallel logic programming, in *Logic Programming: Proceedings of the Tenth International Conference*, MIT Press, 1993. {4}

133. Z. Somogyi, F. Henderson, T. Conway, The execution algorithm of Mercury: an efficient purely declarative logic programming language. Revised version of paper appearing in *ILPS'94 Post-Conference Workshop on Implementation Techniques for Logic Programming Languages.* {8, 18, 63, 69}

134. R. Sundararajan, An abstract interpretation scheme for groundness, freeness and sharing analysis of logic programs, Technical report CIS-TR-91-06. Dept. of Computer and Information Science, University of Oregon, 1991. {24}

135. H. Søndergaard, An application of abstract interpretation of logic programs: Occur check reduction, in *ESOP'86 Proceedings European Symposium on Programming*, LNCS 213, Springer Verlag, 1986. {24}

136. G.L. Steele, *Rabbit: A Compiler for Scheme*, MIT AI Memo 474, Massachusetts Institute of Technology, 1978. {6, 25, 38}

137. L. Sterling, E. Shapiro, *The Art of Prolog*, MIT Press, 1986 {2}

138. P. Tang & P.-C. Yew, Processor Self-Scheduling for Multiple Nested Parallel Loops, in *Proc. 1986 International Conference on Parallel Processing*, August 1986. {91, 109}

139. P. Tarau, M. Boyer, Elementary logic programs, in *Programming Language Implementation and Logic Programming 1990*, LNCS 456, Springer Verlag, 1990. {5, 16, 27}

140. P. Tarau, Low-level issues in implementing a high-performance continuation passing Prolog engine, in *Programming Language Implementation and Logic Programming '92*, LNCS 631, Springer Verlag, 1992. {5, 37}

141. P. Tarau, WAM-optimizations in BinProlog: towards a realistic Continuation Passing Prolog Engine, Technical Report 92-3, Université de Moncton, 1992. {5, 36, 37}

142. A. Taylor, *High Performance Prolog Implementation*, Ph.D. Thesis, Basser Department of Computer Science, Sydney University, 1991. {3, 8, 18, 23, 52, 69}

143. A. Taylor, Parma – bridging the gap between imperative and logic programming, Journal of Logic Programming, Special Issue on High-Performance Implementations. To appear. {3, 23}

144. E. Tick, Memory- and buffer-referencing characteristics of a WAM-based Prolog, *J. Logic Programming*, Vol. 11, pp. 133–162, 1991. {39, 131}

145. E. Tick, C. Banerjee, Performance evaluation of Monaco compiler and runtime kernel, in *Logic Programming: Proceedings of the Tenth International Conference*, MIT Press, 1993. {4, 158}

146. E. Tick, The deevolution of concurrent logic programming languages, Journal of Logic Programming 23(2), 89-124, 1995. {4}

147. B.-M. Tong, H.-F. Leung, Concurrent constraint logic programming on massively parallel SIMD computers, in *Proc. International Symposium on Logic Programming 1993*, MIT Press, 1993. {3, 4}

148. H. Touati, A. Despain, An empirical study of the Warren Abstract Machine, in *IEEE Symp. Logic Programming 1987*, IEEE Press, 1987. {39}

149. H. Touati, T. Hama, A light-weight prolog garbage collector, in *Proceedings of the International Conference on Fifth Generation Computing Systems*, Ohmsha, 1988. {19, 73}

150. A.K. Turk, Compiler optimizations for the WAM, in *Third International Conference on Logic Programming*, LNCS 225, Springer Verlag, 1986. {2}

151. S-Å. Tärnlund, Reform, unpublished manuscript, Computing Science Department, Uppsala University, 1991. {86, 102}

152. K. Ueda, *Guarded Horn clauses*, ICOT Technical Report TR-103, Institute for New Generation Computing Technology, Tokyo, 1985. {4, 21}

153. P. Van Hentenryck, A. Cortesi, B. Le Charlier, Evaluation of the domain `Prop`, Journal of Logic Programming Vol. 23, pp. 237-278, Elsevier, 1995. {24}

154. P.L. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, Ph.D. Thesis, UCB/CSD 90/600, Computer Science Division (EECS), University of California, Berkeley, 1990. {3, 17, 18, 23, 36, 52, 56, 57, 60, 69}

155. P. Van Roy, A. Despain, The benefits of global dataflow analysis for an optimizing Prolog compiler, in *Proc. North American Conference on Logic Programming'90*, MIT Press, 1990. {3, 8, 18, 64, 69, 124}

156. Peter Van Roy, 1983-1993: The wonder years of sequential Prolog implementation, Journal of Logic Programming, 1994:19,20:385-441 {3}

157. A. Voronkov, Logic programming with bounded quantifiers, in *Logic Programming*, LNAI 592, Springer Verlag, 1992. {4, 22}

158. D.H.D. Warren, *Implementing Prolog – compiling predicate logic*, DAI Technical Reports 39-40, Edinburgh University {2}

159. D.H.D. Warren, *An abstract Prolog instruction set*, Report 309, SRI International, 1983. {2, 5, 15, 26, 36, 40, 55, 71, 74, 90, 125, 132}

160. D.H.D. Warren, The Andorra model, presented at Gigalips workshop, University of Manchester, 1988. {4, 21}

161. M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989 {22}

162. R. Yang, *A Parallel Logic Programming Language and Its Implementation*, Ph.D. Thesis, Department of Electrical Engineering, Keio University, Yokohama, 1986. {4, 21}

163. R. Yang, T. Beaumont, I. Dutra, V. Santos Costa, D.H.D. Warren, Performance of the compiler-based Andorra-I system, in *Logic Programming: Proceedings of the Tenth International Conference on Logic Programming*, MIT Press, 1993. {4, 21, 158}

164. N.-F. Zhou, On the scheme for passing arguments in stack frames for Prolog, in *Proc. Eleventh Intl. Conf. on Logic Programming*, MIT Press, 1994. {40}

Variable classification

Variable classification is the basis of Reform compilation, and is further motivated by Millroth [100].

A recursion-parallel predicate p has the following form.

$$
\begin{array}{lcl}
p([], s_2, \ldots, s_n) & \leftarrow & \Upsilon \\
p([X\,|\,Xs], t_2, \ldots, t_n) & \leftarrow & \Phi \wedge p(Xs, t'_2, \ldots, t'_n) \wedge \Psi
\end{array}
$$

Consider the clause head $p(t_1, \ldots, t_n)$ and the recursive call $p(t'_1, \ldots, t'_n)$. The compiler classifies each pair of arguments (t_i, t'_i), for $1 \leq i \leq n$ into one of the following categories.

Class	Head	Rec.call
POSLIST(x)	$[x \mid xs]$	xs
NEGLIST(x)	xs	$[x \mid xs]$
INV(x)	x	x
NONE-NEG(x, y)	x	y

For POSLIST(x) and NEGLIST(x) arguments, we furthermore require that xs does not occur in Φ or Ψ or in other head or recursive call arguments. However, x may occur as POSLIST(x) or NEGLIST(x) classifications for several arguments, or occur in Φ or Ψ. For INV(x) and NONE-NEG(x, y) arguments, we require x and y not to occur elsewhere in the head or recursive call arguments. They may occur in Φ or Ψ.

We will omit the variables from a classification, e.g., writing NONE-NEG rather than NONE-NEG(x, y), when this is obvious from the context or irrelevant to the discussion.

Informally speaking, a POSLIST argument is a list that is traversed by successive recursive calls; a NEGLIST argument is a list built by successive

recursive calls; an INV argument is one that remains invariant over recursive calls, and a NONE-NEG argument is an argument where the head and recursive call arguments are unrelated variables.

Arguments that do not fit the categories are rewritten by breaking out unifications; if the arguments of the rewritten predicate still cannot be categorized, the predicate is rejected by the parallelizer [85].

Code Generator Internals

This chapter gives a brief overview of the Reform Prolog compiler. We describe code generation for recursion-parallel predicates and how the compiler generates code for predicates that are executed by the parallel loops.

B.1 CODE GENERATOR STRUCTURE

The code generation stage of the compiler is a slightly modified WAM compiler, written by the author but based on that described in Carlsson's thesis [26]. Prior to code generation, the analyzer has annotated each variable occurrence in every predicate with its mode, and its locality [85].

There are three classes of predicates handled by the compiler. The last two are unique to Reform Prolog.

- Sequential predicates. These are treated as in a standard WAM compiler.
- Recursion-parallel predicates. These are compiled using a special code generation schema, described in Section B.2.
- Parallel predicates. These are the predicates called by recursion-parallel predicates. The compiler emits code that ensures safeness and attempts to remove the overheads for safeness as thoroughly as possible.

B.2 CODE GENERATION FOR RECURSION-PARALLEL PREDICATES

Recursion-parallel predicates

A recursion-parallel predicate p has the following form.

$$
\begin{array}{lll}
p([], s_2, \ldots, s_n) & \leftarrow & \Upsilon \\
p([X \mid Xs], t_2, \ldots, t_n) & \leftarrow & \Phi \wedge p(Xs, t'_2, \ldots, t'_n) \wedge \Psi
\end{array}
$$

It is compiled into code that (a) builds vectors and execute recurrences (described in Ref. [85, pp. 73-75]) (b) executes all instances of Φ, (c) executes the single recursive call, and (d) executes all instances of Ψ.

Building vectors

The arguments of the predicate are categorized into the four categories defined in Appendix A: POSLIST, NEGLIST, INV or NONE-NEG. These classifications are then used for generating vector code. The vector building instruction set is further described in the main body of the thesis, as well as in Bevemyr's thesis [17].

Class	Instruction
POSLIST(x)	`build_rec_poslist`
	`build_poslist`
	`build_poslist_value`
NEGLIST(x)	`build_neglist`
	`build_neglist_value`
INV(x)	-
NONE-NEG(x, y)	`build_variables`

For POSLIST and NEGLIST, x may have occurred previously or not. If this is not the first occurrence, the `value` versions of the instructions are used.

We assume that the first argument of the predicate is the one used for parallelization. We use `build_rec_poslist` for this argument, rather than `build_poslist`.

B.3 CODE GENERATION FOR PARALLEL PREDICATES

In this section, we use the qualifier "parallel" to denote a construct intended to execute in a parallel setting. For example, a parallel predicate is one intended to be called from a recursion-parallel predicate; a parallel unification is intended to execute in a parallel setting, and so on.

Instruction set

The WAM instruction set is extended by three operations to (conditionally) suspend the computation and three operations to perform locking unifications.

`await_nonvar R.` Suspend until `R` is bound or process is leftmost.

`await_strictly_nonvar R.` Suspend until `R` is bound; if process becomes leftmost, abort with error.

`await_leftmost.` Suspend until process is leftmost.

`lock_and_get_structure f/n,R,T.` Begin locking unification.

`unlock T.` End locking unification.

`global_get_constant c,R.` Atomic unification with constant.

Code generation for parallel unification

Recall that a variable possibly shared between processes is robust, and that a robust variable tested by time dependent tests is fragile. The compiler ensures sequential semantics and binding determinism by emitting the proper form of `await` instruction when fragile variables are accessed. When robust or fragile variables are bound, locking unification must be used. We simplify the exposition a bit by assuming that a unification $T_1 = T_2$ has been simplified into a sequence of normalized unifications on the form $X = f(X_1, \ldots, X_n), X = c$ or $X = Y$.

Unification with a structure. For a unification $X = f(X_1, \ldots, X_n)$, where X has appeared previously and $X_1, \ldots, X_n$ occur the first time in the clause, the compiler uses information on the mode and locality of X to emit the proper instructions. The possible modes are **gnd**, **nv**, **free** or **any**. The possible locality is **local**, **robust** or **fragile**. The process being compiled may at any given time be certainly deterministic (**det**) or possibly nondeterministic (**nondet**). Each predicate is annotated with information when this may change. The other elements of the abstract domain are collapsed to one of these by the annotator.

1. If X has mode **gnd** or **nv**, or locality **local** then a standard WAM unification is emitted.

2. If X has mode **any** and locality **robust**, and the process state is **det**, then the compiler emits:

   ```
   lock_and_get_structure f/n,X,T
   <unify arguments>
   unlock T
   ```

   Note that `T` is a fresh temporary register.

   If the process state is **nondet**, the compiler emits a warning and the code:

```
await_strictly_nonvar X
get_structure f/n, X
...
```

Thus, the engine checks that X is bound when the process is leftmost, or an error is signalled. Otherwise, X is bound and locking unification is not required.

3. If X has mode **any** and locality **fragile**, then check the process determinism.

   - Process is **det**: emit `await_nonvar X`
   - Process is **nondet**: emit `await_strictly_nonvar X`

   This is followed by the code for a locking unification.

4. If X has mode **free**, it can be treated as if **any**. (The compiler can improve safeness checking by taking freeness into account, but we shall not further discuss that topic.)

Note that an alternative solution would be for the compiler to dynamically check determinism. When analysis says the process state is **nondet**, this is a conservative approximation, and compliance with the execution model could be checked at runtime. For example, the engine could signal a binding determinism failure if the binding is shared and actually trailed at runtime, by emitting special instructions that check determinism when a potentially shared variable is bound. This yields some extra flexibility, but may make the system harder to understand.

Unification with constants. When a unification $X = c$ with a constant c is found, we perform the same case analysis as for $X = f(X_1, \ldots, X_n)$. However, we emit `global_get_constant c,X` rather than the locking unification sequence shown above.

General unification. When a general unification $X = Y$ is encountered, the following is done (in the order shown).

1. If X and Y are both **gnd** or **local**, then a normal general unification is emitted.

2. If X or Y is **free** and **local**, then a normal general unification is emitted.

3. If either X or Y is **fragile**, emit `await_leftmost` and go to step 4.

4. If the process state is **nondet**, then binding determinism may be violated. The compiler either rejects the program for parallelization, or may use a version of general unification that checks whether unsafe bindings have been made after the unification. See also below.

5. If the process is **det**, then either or both of X and Y may be shared between processes. Emit the proper version of general unification that performs locking unifications when binding variables in X or Y or both.

At present, the compiler uses a single version of parallel general unification, which uses locking unification throughout both arguments. Also note that one could eliminate `await_leftmost` by writing versions of general unification that suspend appropriately when variables are found. For example, if we unify a fragile term X with a local term Y, then unification could `await_nonvar` whenever it was about to bind a variable in X. This would shift work from the compiler to the engine, and could improve programs with imprecise analysis information. An example of this is shown for the parallel primitive compare/3 below.

Code generation for parallel primitives

The Reform engine has one instruction for parallel primitive operations.

`par_builtin` $S_1, \ldots, S_n$ `p/m` $R_1, \ldots, R_m$ Perform `await_nonvar` on all S_i, then execute primitive $p(R_1, \ldots, R_m)$.

If $n = 0$, the system can reduce the parallel primitive to a sequential primitive. An exception to this rule is when locking is needed to update some argument register R_j.

The compiler keeps a table indexed by primitive name, organized as follows.

Name	Safeness	Suspension
...	...	...
atom/1	[s]	[1]
...	...	...
compare/3	[u,s,s]	[]
...	...	...

Consider the entry for atom/1. The safeness entry means the first argument is not bound by atom/1 (i.e., it is safe), while the suspension entry means the compiler will have to suspend the operation if argument 1 is **fragile**.

For compare/3, the first argument may be bound by the primitive, while the other two arguments are tested. We assume that compare/3 is implemented to suspend appropriately, so the suspension list is empty. In effect, the engine takes care of the problem rather than the compiler.

Code generation for parallel indexing

Parallel indexing is straightforward, since the Reform compiler only uses standard first-argument indexing. If the indexing argument has mode **any** and is **fragile**, the compiler emits `await_nonvar x0` prior to the indexing code. The compiler also uses modes to omit indexing cases that will never occur, e.g., eliminating all non-variable cases if the first-argument mode is **free**.

Performance of Reform Prolog

We provide a performance comparison of Reform Prolog with SICStus Prolog. We also summarize performance measurements of other dependent and-parallel systems, and provide some tentative conclusions.

C.1 REFORM PROLOG VS. SICSTUS PROLOG

We have compared the performance of Reform Prolog with SICStus Prolog 3 (patchlevel 3).

	nc	em	r	r/n	r/em
nrev(900)	230	800	1500	6.52	1.88
match(96)	2550	8120	16640	6.53	2.05
tsp(48)	3250	10060	21040	6.47	2.09
Geo.mean	-	-	-	6.51	2.00

The ‘nc’ column measures SICStus 3 fastcode, the ‘em’ column measures SICStus 3 compactcode and column ‘r’ measures Reform Prolog v.0.7, running one parallel worker. Columns ‘r/nc’ and ‘r/em’ display speedups of native and emulated SICStus versus Reform Prolog.

All timings were measured in milliseconds of runtime on a 4-processor Sparc multiprocessor with 55 MHz processors in normal use. The timing we used was the best of five runs.

As can be seen, emulated SICStus is twice as fast as (emulated) Reform Prolog. We attribute this to the more mature implementation of SICStus Prolog. For example, SICStus Prolog merges adjacent WAM instructions to decrease instruction decoding overhead, which is not done by Reform Prolog.

C.2 REFORM PROLOG VS. OTHER PARALLEL SYSTEMS

Comparisons between Reform Prolog and other parallel logic programming systems is difficult. We attempt to provide an overview of reported performance for some other dependent and-parallel systems, but ask the reader to consider the limitations of this study. Benchmarks, hardware and number of processors used differ, and so these results should be taken as a rough indication of performance, rather than a systematic study. The reported results are a compilation of published results. We use SICStus 2.1 emulated and SICStus 3 emulated as baseline systems.

Andorra-I is an emulator-based implementation of the Andorra principle done at Bristol, written by Yang, Beaumont, Santos Costa, Dûtra and Warren. JAM is an emulator-based Parlog implementation by Jim Crammond. Panda is an emulator-based implementation of Flat Guarded Horn Clauses. Monaco is a KL1 compiler generating optimized native code, written by Evan Tick and his students. Janus is a compiler from the Janus language into sequential C, written by Debray and his students. Penny is an emulator-based parallel AKL system written by Johan Montelius. DASWAM is an emulator-based implementation of dependent and-parallel Prolog, based on extending independent and-parallelism, by Kish Shen.

System	Ratio	Versus	Reference
Reform Prolog	appx. 1	SICStus 2.1	J. Bevemyr
Andorra-I	3.7	SICStus 2.1	[163]
Strand	appx. 1	SICStus 2.1	[145]
JAM	1.6	SICStus 2.1	[145]
Panda	2.5	SICStus 2.1	[145]
Monaco	0.62	SICStus 2.1	[145]
Janus	0.27	SICStus 2.1	[145]
SICStus 3, nc	0.31	SICStus 3	
Reform Prolog	2.0	SICStus 3	
Penny	2.5	SICStus 3	J. Montelius
DASWAM	2-4	SICStus 3	[128]

Our tentative conclusions from these numbers are the following.

- Of the emulator based systems, Reform Prolog appears to be the fastest, with DASWAM coming close in some cases. We believe this is due to the frequent context switches and process spawning done by the implicitly concurrent languages (e.g., Andorra-I, Penny, JAM, Panda). In short, optimizing concurrent threads is less well-understood than optimizing sequential computations at this point of time. As can be seen, more recent concurrent systems (e.g., Penny) are approaching the speed of Prolog.

- Generating native code appears to yield a factor 4-8 performance improvement over emulation for the implicitly concurrent languages (Janus, Monaco) as well as Prolog (SICStus 3, native). Note that Janus is a sequential implementation, while Monaco is parallel.

Parallel Overheads of Reform Prolog

We provide a brief overview and discussion of the parallel time and space overheads of Reform Prolog.

There are three main sources of time overhead for parallelization in Reform Prolog: sequential overheads, loop startup overhead and per-process overhead. Space overheads are present in that vector lists allocate some variables on the heap rather than the stack.

We estimate the cost of parallel operations by counting the number of load-store instructions executed. We assume that sufficient registers are available for temporaries, that conditional branches perform a test and a branch in one instruction, and that some operations work with shared memory. The shared memory is organized as a number of coherent caches and a main memory. When a location is updated in a cache, all other copies (if any) are invalidated. When a location is read, it is pulled into the appropriate cache.

The actual algorithms used in the Reform engine differ from the algorithm outlines described here, since the former were written for an emulator with portability between several architectures in mind. However, we have based our discussion on the current implementations.

Sequential overheads

The Reform engine uses timestamps to manage parallel trailing. This is represented as having special *unbound cells* that hold an unbound tag and a timestamp value. As compared to the WAM, there are the following overheads.

An extra tag is required. Unbound variables are tagged in two ways, depending on whether they are references or not.

Dereferencing changes. The dereferencing algorithm traverses a chain of reference-tagged cells until a non-reference is found. (Not necessarily an overhead.)

Extra register. The timestamp value is stored in a (virtual or real) register.

Note that writing unbound variables may be faster than in WAM, since there is no need to generate a self-pointer for every cell. Instead, one can simply copy the timestamp into n words.

Extra choicepoint field. The timestamp value is stored in an extra choicepoint field.

Copying variables may require more work. When an unbound cell is copied, the address of the cell must be retained. Other cells can be copied normally. This requires a test and an extra register, unless the appropriate static information is available.

Bevemyr has measured the overhead to be in the range of 5% for the current implementation of Reform Prolog. Static analysis could reduce overheads further (e.g., copying may be strength-reduced when the mode is known).

Loop startup overheads

Loop startup overheads include (a) the cost of building vector lists and (b) the cost of starting and stopping the workers at the beginning and end of the parallel loop.

When building vector lists, there are two cases. For the input recursion POSLIST argument, the list is traversed and the corresponding vector list is built as each element is accessed. For other POSLIST arguments, the recursion size is known, which means a smaller overhead. NEGLIST and NONE-NEG arguments never need traversal, but are created using the second method below.

Traversing the input recursion list x is done by traversing the list and copying it to consecutive cells. We assume there is a pointer to the previous tail of the list, *prev*, which is filled in when a new cell is allocated. A sketch of the inner loop is shown below.

1. Dereference x.
2. Select tag of x. (1 and-instruction; we assume it can be done by masking with a small constant)
3. If x is not a list pointer, then traversal is done. (1 conditional branch)

4. Allocate 2 cells. (1 conditional branch + 1 add)
5. Load car of x.
6. Load cdr of x into x.
7. Store car of x in cell 1.
8. Move listptr(cell_1) to t.
9. Store t in $prev$.
10. Set $prev$ to cell 2.
11. Increment n, the list length.
12. Go to beginning of loop.

The cost for traversing the input list is thus 5 alu-operations, 2 loads, 2 stores, 2 conditional branches and 1 unconditional branch per list element traversed. Furthermore, a dereferencing operation is required; the precise timing of this operation depends on whether there is a pointer chain to be traversed. A typical dereferencing might execute 2-5 instructions. Static analysis can remove the dereferencing operation entirely in some situations.

POSLIST arguments may be instantiated or not. If a POSLIST argument is unbound, the vector list is built by the method below. If it is bound, then the bound portion of the spine must be traversed as for a recursion list (above). Should this spine end in an unbound variable, the remaining elements are filled in using the method below [17].

Building a non-recursion vector list is done by the following stylized inner loop. Since we know n, no GC test is needed in the inner loop and the $prev$ fields can be built immediately. We assume a to point to the list block to be filled in, that b points to a vector list to be copied, and that there are 4 bytes per word.

1. Load x from b.
2. Store x in a.
3. Move the tagged list pointer listptr$(a + 8)$ to t.
4. Store t in $a + 4$.
5. Decrement n.
6. Add 8 to a.
7. Add 8 to b.

8. If $n > 0$, go to beginning of loop.

The cost is estimated to 1 load, 2 stores, 4 arithmetic operations and 1 conditional branch.

Note that this loop is very amenable to loop unrolling. Unrolling by k eliminates $k-1$ backward branches, and subsequent constant propagation can eliminate most arithmetic operations. The cost then approaches 1 load and 2 stores in the limit.

There are a number of variations on this basic theme. When executing non-VALUE build instructions, an unbound variable is written in every car cell, instead of x.

The cost of starting and stopping workers is highly dependent on the implementations of barriers and worker suspension and on the operating system, and we will not discuss it further.

Discussion The cost of traversing the input list can be substantially reduced by loop unrolling. In particular, it reduces the number of backward branches and garbage collection tests. A loop that is unrolled by n removes $n-1$ branches and $n-1$ garbage collection tests.

For other arguments, space for the entire vector list can be allocated at once. Again, loop unrolling helps, by removing backward branches.

Per-process overheads

The per-process overheads are broken down into the cost of scheduling a recursion level onto a worker, the costs of locking bindings versus those of standard bindings, and the cost of suspension.

Scheduling Scheduling a recursion level for execution has two subcosts: first, a worker must be assigned the recursion level; second, the worker must load the arguments of the recursion level. We assume a loop limit counter is set, and that there is a local iteration counter per worker. The loop limit is read-only during the parallel loop. We also require a *worker array*, which is further discussed in Section D.

Workers are assigned recursion levels statically or dynamically. Static assignment simply gives worker i the recursion levels $i, i+w, i+2w, \ldots$.

1. Load the local iteration counter and loop limit (if not in a register).

2. Increment iteration counter.

3. Load address of worker array.

4. Store the current iteration in the worker array (explained in Section D). This is a shared operation.
5. If iteration counter exceeds loop limit, the worker goes to sleep.
6. Otherwise, store iteration counter and begin the next recursion level.

We estimate the time for these operations as 3 loads, 1 add, 1 conditional branch and 2 stores. (The limit test + start of next level can be combined.) This yields a sum of 7 instructions, of which one is a shared memory operations. If iteration counter, loop limit and worker array address are found in registers, 3 loads and 1 store can be deleted.

Dynamic assignment is slightly more complex. A shared loop iteration counter is used, which requires locking and updating it. The counter is locked by exchanging the iteration counter with a lock value, which is an invalid iteration count. The lock is released by writing the next iteration to be scheduled.

1. Move a (constant) lock value into a register r.
2. Exchange r with the shared loop iteration counter.
3. If r now contains the lock value, retry the lock. (Some other worker is locking the iteration counter.)
4. Increment iteration counter.
5. Store iteration counter into the shared cell.
6. Load the address of the worker array.
7. Store the current iteration in the worker array (explained in Section D).
8. Load loop limit.
9. If iteration counter exceeds loop limit, worker goes to sleep.

We estimate the common case cost as 2 alu operations, 1 exchange, 2 conditional branches, 2 stores, and 2 loads. A total of 10 instructions, of which 3 work with shared memory. If the loop limit and worker array address are found in a register, 2 loads can be deleted.

Loading the vector list arguments is done as follows, per argument.

1. Load local iteration counter.

2. Load the base address of the vector.
3. Multiply iteration counter by 2*sizeof(cell), usually 8 or 16 (thus, use a shift instruction).
4. Load the value at the indicated address (base+offset).

The cost is estimated as 3 loads and 1 shift. If the iteration counter and base address is found in registers, 1 shift + 1 load is required. Both loads are assumed to be shared. Note that some instruction sets have addressing modes that merge the shift and last load.

There are instruction set variations on the theme above; e.g., loading the tail of a vector list requires adding sizeof(cell) as an offset.

Finally, loading a non-vector list argument (i.e., an INV argument), costs less.

1. Load the address of the INV argument.
2. Load the value at the address.

This requires 2 loads, both of which may be shared.

Locking bindings A locking binding is done as follows. We want to bind the variable to an already constructed term t (i.e., t is a tagged pointer or immediate). The algorithm is taken from Bevemyr [17, p.17].

1. Move t into register r.
2. Exchange r with the variable cell.
3. Select the tag of r. (We assume this can be done by masking with a small constant.)
4. If the tag of r is not that of an unbound variable, then perform a general unification.

The common case cost is estimated to 2 alu operations, 1 exchange and 1 conditional branch. Note that general unification is almost never required in the programs we have measured. Furthermore, note that binding determinism ensures that no trailing is required.

The Reform compiler generates the code:

```
lock_and_get_structure f/3, X0, T
unify_variable X1
unify_variable X2
unify_variable X3
unlock T
```

In a native code setting, we can move the atomic binding into the unbound case.

```
     t0 = deref(X0);
     t1 = tag_of(t0);
     if (t1 == UNB) goto unb;
     <load arguments>
done: <rest of code>
     ...
unb: <build f(X1,X2,X3)>
     <locking binding>
     goto done;
```

Suspension Generally, a worker suspends waiting for a value to become bound, or waiting to become leftmost. Leftmostness checking is currently implemented as follows: the engine allocates an array of counters, one per worker. When a worker schedules a new recursion level, it stores the new recursion level number in the array. A worker checks whether it is leftmost by traversing the array. If its own counter is the least of all counters, the worker is leftmost.

This yields the following algorithm for suspending on value x, taken from Bevemyr [17]. Note that we assume unbound variable cells and references to have different tags.

1. Load value of x. (1 load)
2. Select tag of x. (1 and-instruction)
3. If x is not unbound, then exit suspension loop. (If x is a reference cell, follow the reference chain and resuspend if it ends in an unbound cell.) (1 conditional branch)
4. Load current iteration counter c. (1 load)
5. Load w, the number of workers. (If this is a static constant, only a move is needed.) (1 load)
6. Set $i = w$, where w is the number of workers. (1 alu)

Operation	Est. overhead (shared)
Traversing rec.list	(12 + deref) per element
Vector list creation	8 per element
Static scheduling	7 (1)
Dynamic scheduling	10 (3)
Loading vector argument	4(2)
Loading non-vector arg.	2(2)
Suspension	$6 + 4w(w + 1)$ per iteration
Locking binding	4(1)

Figure D.1: Estimated overheads for parallelism, in number of instructions. The number of shared memory operations is shown in parentheses.

(a) Load $a[i]$. (1 load)

(b) If $a[i] < c$, then go to Item 1. (1 conditional branch)

(c) Decrement i. (1 alu)

(d) If $i \neq 0$, go to Item 6.(a). (1 conditional branch)

We estimate the cost of the suspension loop to be 3 loads, 2 alu operations and 1 conditional branch, plus the inner loop, which requires 1 load, 2 conditional branches and 1 add per loop iteration. One load is shared in the outer loop, and one load is shared in the inner loop. The time for one iteration of the outer suspension loop is then at worst $6 + 4w$ instructions, of which $w + 1$ operations are shared. We can call this the 'reaction time' of a suspended process.

D.1 SUMMARY AND DISCUSSION

A summary of the estimated overheads is given in Figure D.1. From these numbers we can see the following.

- Due to loop startup overheads, there is a minimum efficient granularity of recursion levels. Given i vector list arguments, 1 recursion argument and j INV arguments, the overhead is estimated to $12 + 12i + 2j$ plus scheduling of 7 or 10 instructions and a dereferencing operation.

  Implementations should increase the process size when this overhead becomes significant. A simple method is to unroll the loop and compile several recursion levels as a single process.

- Locking binding is not a problem, unless workers compete intensely for a substantial number of bindings, or the cost to transfer data through the machine is extremely high.

If there is little competition for a binding, the binding cost is small. In essence, the cost is that of broadcasting the binding to all workers suspended on it. In the current implementation and for the current benchmarks, about one locking binding in 10,000 does not succeed immediately (J. Bevemyr, personal communication).

- Suspension determines how quickly a worker can start executing once its value has become bound or it becomes leftmost. The time is essentially limited by the number of workers, w. Thus, if there are many workers, an alternative design may be more suitable.

  It is at present unclear whether several counters should be stored in a single cache line or not. The disadvantage is that writing a counter invalidates several counters from other caches; the advantage is that reading one counter prefetches all other counters in the same cache line.

- The per-process overhead appears to be largely determined by the number and length of suspensions per process. If a process suspends for long periods, then parallel overhead will naturally increase. But less obviously, frequent suspension checks will also reduce performance, even if the process never actually suspends.

  Thus, the compiler should reduce the number of suspension operations as far as possible. In the end, the algorithm requires a minimum number of suspensions. To go beyond that point, the program must be rewritten to reduce dependences between recursion levels.

D.2 SPACE OVERHEADS

Allocating vector lists leads to some space overheads. First of all, the recursion list is copied to a vector. Second, instantiated parts of other POSLIST arguments are copied as well. Third, NONE-NEG variables, which might have been allocated on the stack or even in registers (e.g., as output variables), appear on the heap. NEGLIST and unbound POSLIST arguments will be built whether running in parallel or sequentially, and so do not impose any overhead.

With a generational garbage collector, we believe the overheads of reclaiming dead vectors will be small. However, the transient space overhead of vector lists still remains. Should this space overhead prove too costly, we can trade speed for space.

First, Reform Prolog at present represents NONE-NEG vectors as vector lists. An alternative is to represent NONE-NEG variables as a simple vector, which halves the space overhead. If this is still too much, the parallel loop can be unrolled to reduce the number of NONE-NEG variables. Unrolling

by k divides the number of NONE-NEG variables by k. These two options would require extending the compiler as well as runtime system.

Second, vectors are created so that workers can quickly locate all the k'th elements (in order to start recursion level k). Currently, the system builds vectors that allow constant time access to all elements k. An alternative is to not copy the list spines of instantiated POSLIST arguments at all (including the recursion list). Consider a worker that has just run recursion level k, and now is starting $k + c, c > 0$. List element $k + c$ can be located by traversing the list c steps, starting from the current element k. If this approach is used, all space overhead except for NONE-NEG variables is eliminated. The cost is slower scheduling of a recursion level. This option can be implemented inside the current engine, possibly as directed by the compiler.

Finally, it may be possible to reuse already created vectors and vector lengths by compiler transformations. Iterative recursion-parallel computations (such as the **ga** benchmark) may be particularly amenable to such an approach. This option is mostly compiler driven.

Computing Science Department
Uppsala University
Box 311, S–751 05 Uppsala, Sweden

Uppsala Theses in Computer Science 26
ISSN 0283-359X
ISBN 91-506-1181-X