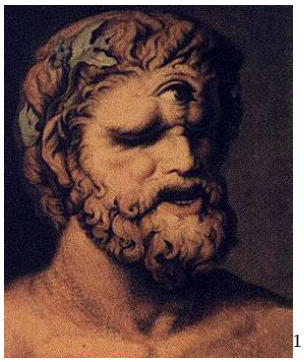
CICLOPS 2012

12th International Colloquium on Implementation of Constraint and LOgic Programming Systems

Workshop of the 28th International Conference on Logic Programming

4th September 2012
Budapest, Hungary

 [1]

Editors

Nicos Angelopoulos
Roberto Bagnara

Published by the ICLP committee

[1] Polyphemus, by Johann Heinrich Wilhelm Tischbein, 1802

Preface

This volume contains the papers presented at CICLOPS'12: 12th International Colloquium on Implementation of Constraint and LOgic Programming Systems held on September 3, 2012 in Budapest.

The program includes 1 invited talk. There were 9 papers accepted, each reviewed by 3 reviewers.

CICLOPS'12 continues a tradition of successful workshops on Implementations of Logic Programming Systems, previously held in Budapest (1993) and Ithaca (1994), the Compulog Net workshops on Parallelism and Implementation Technologies held in Madrid (1993 and 1994), Utrecht (1995) and Bonn (1996), the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages held in Port Jefferson (1997), Manchester (1998), Las Cruces (1999), and London (2000), and more recently the Colloquium on Implementation of Constraint and LOgic Programming Systems in Paphos (2001), Copenhagen (2002), Mumbai (2003), Saint Malo (2004), Sitges (2005), Seattle (2006), Porto (2007), Udine (2008), Pasadena (2009), Edinburgh (2010) - together with WLPE, Lexington (2011).

We would like to thank all the authors of submitted papers, the programme committee members, the ICLP 2012 organisers for support and the EasyChair conference management team.

August 4, 2012

Nicos Angelopoulos
Roberto Bagnara

Program Committee

Nicos Angelopoulos	Netherlands Cancer Institute, The Netherlands
Roberto Bagnara	University of Parma, Italy
Miguel Calejo	Declarativa, Portugal
Mats Carlsson	SICS, Sweden
Daniel Diaz	University of Paris, France
Rémy Haemmerlé	Universidad Politécnica de Madrid, Spain
Günter Kniesel	University of Bonn, Germany
Paulo Moura	CRACS - INESC Porto and University of Beira Interior, Portugal
Ricardo Rocha	University of Porto, Portugal
Guido Tack	Monash University, Australia
Paul Tarau	University of North Texas, USA
Markus Triska	Vienna University of Technology, Austria
Jan Wielemaker	SWI Prolog, Free University of Amsterdam, The Netherlands
Neng-Fa Zhou	Brooklyn College, USA

Author Index

Table of Contents

Tor: Modular Search in Prolog

Thom Schrijvers

University of Ghent, Belgium
tom.schrijvers@ugent.be

Abstract. Prolog has a depth-first procedural semantics. Unfortunately, this procedural semantics is ineffective for many programs. Instead, to compute useful solutions, it is necessary to modify the search method that explores the alternative execution branches. Tor is a well-defined hook into Prolog disjunction that provides this ability. It is general enough to mimic search-modifying predicates like ECLiPSe's search/6. Tor's light-weight library-based approach puts these search methods at the fingertips of every programmer in any program. Moreover, Tor supports modular composition of search methods and other hooks, and obtains efficiency through program transformation. The Tor library is currently available for SWI-Prolog and B-Prolog.

Toward a Dynamic Programming Solution for the 4-peg Tower of Hanoi Problem with Configurations

Neng-Fa Zhou and Jonathan Fruhman

Department of Computer and Information Science, Brooklyn College
The City University of New York
New York, NY 11210-2889, USA

Abstract. The Frame-Stewart algorithm for the 4-peg variant of the Tower of Hanoi, introduced in 1941, partitions disks into intermediate towers before moving the remaining disks to their destination. Algorithms that partition the disks have not been proven to be optimal, although they have been verified for up to 30 disks. This paper presents a dynamic programming approach to this algorithm, using tabling in B-Prolog. This study uses a variation of the problem, involving configurations of disks, in order to contrast the tabling approach with the approaches utilized by other solvers. A comparison of different partitioning locations for the Frame-Stewart algorithm indicates that, although certain partitions are optimal for the classic problem, they need to be modified for certain configurations, and that random configurations might require an entirely new algorithm.

1 Introduction

The classic 4-peg Tower of Hanoi problem is posed as follows:

Given: Four pegs, and n disks of differing sizes stacked in increasing size order on one of the pegs, with the smallest disk on top.

Goal: Stack the n disks on a different one of the four pegs, using the following rules:

1. Only one disk can be moved at a time.
2. Only the top disk on any peg can be moved.
3. Larger disks cannot be stacked above smaller disks [15].

A number of authors have created algorithms to try to solve both this puzzle and the related puzzle involving an arbitrary number of k pegs. Frame and Stewart provided the first documented algorithms for the k-peg problem [4] [15]. Their algorithms involved partitioning the disks into intermediate sub-towers. Since the time that Frame and Stewart published their results, others have analyzed algorithms in order to either attempt to solve the 4-peg and k-peg Tower of Hanoi puzzles using a minimal number of moves, or to try to prove the optimality of the number of moves generated by Frame's and Stewart's algorithms. Section 2 discusses the Frame-Stewart algorithm.

The goal of this paper is to make steps toward an efficient dynamic programming solution for the 4-peg tower of Hanoi puzzle. The solution is presented using the B-Prolog

programming language [24], and uses tabling, a technique similar to pattern databases [7], in order to decrease the number of necessary computations. The program will be *non-deterministic*, in order to extend the Frame-Stewart algorithm to variants of the 4-peg problem that involve disk configurations. In order to reduce the non-determinism, this paper examines different partition locations for the intermediate sub-towers, in order to find the most efficient one to use for the configuration problem. Although there is no guarantee that the 4-peg Frame-Stewart algorithm is optimal [10], the dynamic programming solution utilizes this algorithm, as it has been proven to generate the optimal solution for up to 30 disks [7].

In Sect. 3, the configuration problem is examined, as posed by the 2011 ASP Competition [19]. The focus is team BPSolver's program [28], which uses Prolog together with tabling [27]. BPSolver's results are compared with those of the other teams: team Fast Downward, which used PDDL, and teams EZCSP, IDP, Potassco, and Aclasp, which used SAT solvers and grounders[1]. The comparison will concentrate on Potassco's program, which had the second-best performance for the Hanoi benchmarks. Sect. 3.3 compares dynamic programming approaches for splitting the problem into subproblems, finding one that seems optimal.

Section 4.1 studies the seemingly optimal solution when applied to random configurations that do not extend from Frame's and Stewart's algorithms. Then, in Sect. 4.2, the dynamic programming approaches are applied to the classic problem for up to 30 disks, getting a different result than Sect. 3.3. Finally, Sect. 5 describes alternative approaches to solving the 4-peg and *k*-peg problems.

2 The Frame-Stewart Algorithm

When the *k*-peg Hanoi problem was posed in 1941, two authors provided solutions. One algorithm, provided by J. S. Frame, is iterative. The other, written by B. M. Stewart, the proposer, is recursive. Both authors claim that their algorithms generate the minimum number of moves, while Frame admits that other methods might also be able to provide the same minimum number.

Following is Stewart's algorithm for the 4-peg problem:

1. Move the Mid topmost disks to another, intermediate, peg (which is not the destination peg), using all 4 pegs.
2. Move the $n - Mid$ remaining disks to the destination peg using the 3 remaining pegs.
3. Move the Mid disks from their current peg to the destination peg, using all 4 pegs [15].

Frame's algorithm is similar to Stewart's. According to Frame, for the *k*-peg problem, a series of towers needs to be created. The smallest tower will consist of the largest disks, and the largest tower will consist of the smallest disks. Only the largest disk remains on the start peg. Once the towers are created, the largest disk is moved directly from the start peg to the destination peg. Then, the tower of the next-largest disks can be

[1] https://www.mat.unical.it/aspcomp2011/Participants/

moved to the destination peg using three pegs. Each intermediate tower is then moved to the destination peg using one more empty peg than the previous tower [4].

If the goal is to minimize the number of disk movements, then this algorithm presents two problems. One problem is to find the optimal number of disks, Mid, to place on the intermediate peg. This paper examines the partitioning issue in Sect. 3.3, using Stewart's recursive algorithm. The other issue is that Frame and Stewart do not provide evidence that an optimal aglorithm involves the creation of intermediate subtowers [10]. However, as mentioned above, since Korf has verified the optimality of this algorithm for 4 pegs and up to 30 disks [7], the Frame-Stewart algorithm is a good estimate.

3 The ASP Competition Problem

3.1 The 4-Peg Problem with Configurations

One of the ASP Competition's problems modifies the 4-peg puzzle to include configurations. This problem follows the same rules as the classic problem, but differs in regard to how the puzzle is initially arranged, and regarding the goal of the puzzle. Instead of all the disks beginning and ending on the same peg, this problem provides two *configurations*. The first configuration is the *start state*, a distribution of the disks over any combination of all four pegs. The second configuration is the *goal state*, a different distribution of the disks over the four pegs. These configurations were created based on the moves that the Frame-Stewart algorithm would perform. The problem is to generate the disk moves needed to get the disks from the first configuration to the second. In addition, there is a bound on the number of moves that could be generated [19].

The BPSolver team used B-Prolog to generate solutions for the given configurations. The following is the relevant code:

```
:-table plan4(+,+,+,-,min).
plan4(N,_CState,_GState,Plan,Len):-N=:=0,!,Plan=[],Len=0.
plan4(N,CState,GState,Plan,Len):-
   remove_largest_disk_if_in_place(
           N,CState,GState,CState1,GState1),!,
   N1 is N-1,
   plan4(N1,CState1,GState1,Plan,Len).
plan4(N,CState,GState,Plan,Len):-
   % split disks into two groups
   partition_disks(N,CState,GState,ItState,Mid,Peg),
   % sub-problem1
   remove_larger_disks(CState,Mid,CState1),
   plan4(Mid,CState1,ItState,Plan1,Len1),
   % sub-problem2
   remove_smaller_or_equal_disks(CState,Mid,CState2),
   remove_smaller_or_equal_disks(GState,Mid,GState2),
   N1 is N-Mid,
   plan3(N1,CState2,GState2,Peg,Plan2,Len2),
```

```
% sub-problem3
remove_larger_disks(GState,Mid,GState1),
plan4(Mid,ItState,GState1,Plan3,Len3),
%
append(Plan1,Plan2,Plan3,Plan),
Len is Len1+Len2+Len3.
```

Instead of focusing on the given upper-bound of moves, this program attempts to find the *smallest* number of moves required to get from the start configuration to the goal configuration. Each configuration is represented by a single state, consisting of four lists. Each list represents a single peg, and stores the disks that are currently located on the peg. This is an improvement on the BPSolver team's prior attempts to represent configurations using Boolean expressions. Prolog lends itself to list operations, and the number of disks on a peg is simply the length of the list.

The predicate `plan4` is defined with three clauses. The first clause is the termination condition. When the number of disks to be removed is zero, the problem has been solved. The second clause determines whether the current largest disk is currently in place. If so, it simplifies the problem by logically removing the disk.

The third clause uses a modified form of Stewart's approach to solve the configuration problem. The disks are split into two sub-groups, one of which is placed in an intermediate tower. The problem is separated into three sub-problems:

1. The smallest `Mid` disks are placed on an intermediate peg, `Peg`, by calling `plan4` recursively.
2. The larger disks are moved from their current pegs to their destination positions by using the deterministic 3-peg algorithm on all of the pegs except for `Peg`[2].
3. The small disks in the sub-tower are moved from the intermediate peg to their destination positions by calling the `plan4` predicate recursively [27].

Another important line of code is `:-table plan4(+,+,+,-,min)`. This uses *tabling* to store information about each state in memory. The purpose of tabling is to store answers to sub-goals, and to utilize the answers for future variant sub-goals. This is a useful tool for dynamic programming, which reuses solutions to overlapping sub-problems. B-Prolog uses linear tabling, lets variant sub-goals share answers, and uses the local, or lazy, strategy to return answers [25]. The most recent version of B-Prolog replies on hash-consing to let tabled subgoals and answers share ground structed terms [26].

There are two benefits to using tabling. The first benefit is that tabling prevents infinite loops. Once a state is visited, it should not be revisited. If states are visited more than once, the program could be stuck cycling between multiple states, possibly by just repeatedly moving a single disk between two pegs. By storing states in memory, the program can check the table to see if a state has already been encountered. The other benefit of using tabling is that it reduces the number of calculations. Once the program knows how to move p disks between two pegs, it can check the table to determine how

[2] The 3-peg Tower of Hanoi is deterministic, as it has been proven to have a minimum solution of $2^n - 1$ moves for n disks [1].

to move any other set of p disks between any two pegs. Sub-problems are represented in such a way that the same problem has the same representation and can share answers through tabling. It does not matter what the sizes of the p disks are, nor does it matter which pegs are being used as the current start and destination pegs, assuming the intermediate and destination pegs are logically empty, meaning that they do not contain disks smaller than the largest one being moved. This decreases the number of operations used during the recursive calls, and is helpful when backtracking to test a different solution.

The line `:-table plan4(+,+,+,-,min)` goes together with the line defining the predicate `plan4(N,CState,GState,Plan,Len).` A plus-sign (+) indicates that the corresponding arguments (`N` - the number of disks, `CState` - the current state, and `GState` - the goal state) are input. A minus-sign (−) indicates that the corresponding arguments (`Plan` - the sequence of disk moves) are output. The last part, `min` indicates that `Len`, the length of the plan of disk moves, should be minimized. By minimizing the plan length, the number of moves will clearly be within the bounds provided in the given problems; otherwise, the problems would be unsolvable.

Since the predicate `plan4` is non-deterministic, it presents a few interesting issues. Like the classic algorithm, this program must determine optimal sizes of the disk subtowers. A new problem that arises is that the program must determine which peg to use to store each sub-tower. In the classic 4-peg problem, there is one peg that never has to be used to store a sub-tower [9] [20]. However, the same is not true for the configuration problem. The start and destination pegs can change for each disk, meaning that any peg might need to be used to store a sub-tower. For further discussion of these issues, see Sect. 3.3.

Tabling is also used for `plan3`, defined below for the 3-peg problem.

```
:-table plan3(+,+,+,+,-,min).
plan3(0,_CState,_GState,_UnusedPeg,Plan,Len):-!,Plan=[],Len=0.
plan3(N,CState,GState,UnusedPeg,Plan,Len):-
   remove_largest_disk_if_in_place(N,CState,
                        GState,CState1,GState1),!,
   N1 is N-1,
   plan3(N1,CState1,GState1,UnusedPeg,Plan,Len).
plan3(1,CState,GState,_UnusedPeg,Plan,Len):-!,
   Plan=[(Peg1,Peg2)],Len=1,
   btm_disk_on_peg(1,CState,Peg1),
   btm_disk_on_peg(1,GState,Peg2).
plan3(N,CState,GState,UnusedPeg,Plan,Len):-
   btm_disk_on_peg(N,CState,Peg1),
   btm_disk_on_peg(N,GState,Peg2),
   other_two_pegs(Peg1,Peg2,Peg3,Peg4),
   (UnusedPeg==Peg3->TmpPeg=Peg4;TmpPeg=Peg3),
   N1 is N-1,
   remove_bottom_disk(CState,Peg1,CState1),
   ItState=s(_,_,_,_),
   Tower @= [I : I in N1..(-1)..1],
```

```
foreach(I in 1..4,
        (I==TmpPeg->arg(I,ItState,Tower);
         arg(I,ItState,[])
        )
),
plan3(N1,CState1,ItState,UnusedPeg,Plan1,Len1),
remove_bottom_disk(GState,Peg2,GState1),
plan3(N1,ItState,GState1,UnusedPeg,Plan2,Len2),
append(Plan1,[(Peg1,Peg2)|Plan2],Plan),
Len is Len1+Len2+1.
```

Like `plan4`, the predicate `plan3` has a clause for the termination condition and a clause for reducing the problem when the latest disk is in place. When the number of disks to be moved is one, it takes one step to solve it. Otherwise, the problem is divided into three sub-tasks: the first sub-task is to move N-1 disks except for the largest one from the current peg to a temporary peg; the second sub-task is to move the largest disk to the destination peg; the third task is to move the N-1 disks from the temporary peg to the destination peg. Unlike `plan4`, `plan3` is deterministic.

3.2 Competition Results

The ASP Competition had six participants, including BPSolver[3]. FastDownward used Planning Domain Definition Language to represent the problems, and utilized A* search with the selective-max and landmark-cut heuristics, using the input to bound the maximum solution length, in order to solve the Hanoi problem. The IDP team described the problems by using First Order Logic with Inductive Definitions, translated the problems into Extended Conjunctive Normal Form by using the Gidl grounder, and solved the problems with the MINISAT(ID) solver. The remaining three teams solved the Hanoi problem by using the grounder Gringo, which translates input programs into equivalent, variable-free programs, and the solver Clasp, which focuses on answer set programming together with nogood learning, checking for violated constraints. While EZCSP ran Clasp on its default settings, Aclasp utilized a modified restart strategy, which, as described by the team, "depends on the average decision-level on which conflicts occurred." In order to solve the Hanoi problem, the final team, Potassco, specified that Clasp should use the Variable State Independent Decaying Sum decision heuristic, and limited the amount of preprocessing and the initial database size.

The ASP Competition programs were graded based on two criteria: the correctness of the solution and the amount of time that it took for each program to find the solution. The competition organizers had sixty instances[4] that could have been used to test the Hanoi solvers. Only fifteen of those instances were actually used to grade the

[3] The following implementation details, and the participants' programs, can be obtained at the team description pages, located at https://www.mat.unical.it/aspcomp2011/Participants/. Further details can be found at the solvers' websites, as listed in the teams' descriptions online.

[4] `http://www.mat.unical.it/aspcomp2011/files/HanoiTower/hanoi_tower-full_package.zip`

solvers. Based on those fifteen instances, the BPSolver team scored highest, followed by Potassco, AClasp, and IDP. EZCSP and Fast Downward tied for the lowest score.

BPSolver's program was the only one that actively used the Frame-Stewart algorithm to limit the size of the search space. Every other program defined legal moves for the tower of Hanoi problem, and used the definition as input to their solvers. Therefore, the other teams' scores were a result of the solvers they used and the heuristics that the solvers employed to trim the search space. The active use of the Frame-Stewart algorithm contributed to the speed of BPSolver's program. Although some of the other teams used heuristics, and added rules in order to improve the search, such as EZCSP specifying that no disk should be moved twice in a row, they still had a larger search space. As will be shown in Sect. 3.3, BPSolver used a mathematical function to partition the disks, which limited the possible moves at any given time. In addition, tabling reduced the number of repeated calculations, causing BPSolver's program to run faster than all the other programs.

In the competition, BPSolver's Hanoi program scored 94, while Potassco's scored 81. However, an analysis of their performances on all 60 instances, summarized in Table 1, shows that the margin would not be as large. BPSolver still scores higher than Potassco, but that is primarily due to the speed of B-Prolog, as opposed to the correctness of the program.

Table 1: Comparing B-Prolog's original program with Potassco's program

Sys	AVG Seconds	MAX Seconds	Num Solved	SSolve	STime	Score
BPSolver	0.022883333	0.047	54	45	45	90
Potassco	10.61766667	102.867	60	50	37	87

BPSolver's program printed a result of "UNKNOWN" for 6 of the instances, while Potassco's was able to solve all instances. Four of those six instances are a result of an error[5], in which the program printed "UNKNOWN" after finding a solution. Only one of these instances was run during the competition. The remaining two unsolved instances, which are illustrated in Fig. 1, and which were not run during the competition, are indicative of a larger problem. Team BPSolver mistakenly believed that there was a known optimal guide for partitioning the disks when creating intermediate sub-towers [27]. The B-Prolog program's partitioning guide was not always able to find an optimal solution.

3.3 Partitions

There have been a number of different estimates for determining the optimal size of the intermediate sub-tower when given n disks.

[5] When a sub-tower is built, it should be built on a logically empty peg. The original program did not test if the peg was empty, causing the error.

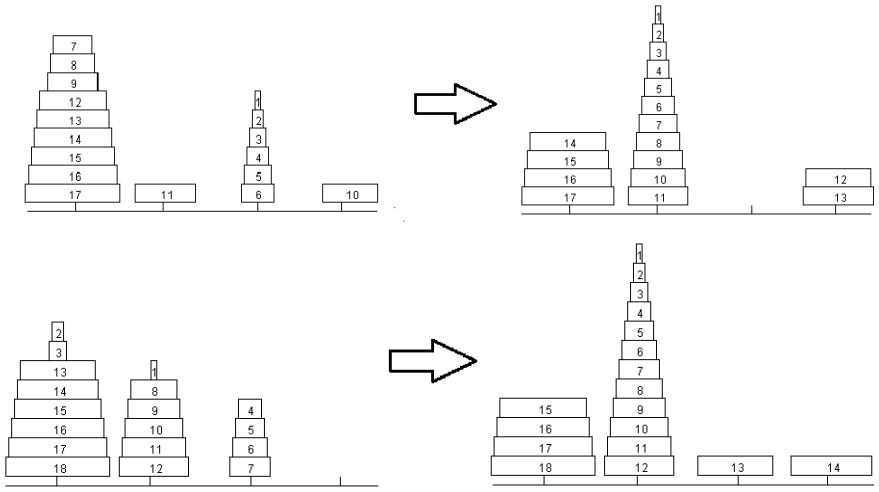


Fig. 1: Instances 17 and 22, the unsolved instances

For the ASP competition, BPSolver used the following code to determine the optimal partition location:

```
partition_disks(N,CState,GState,ItState,Mid,Peg) :-
   btm_disk_on_peg(N,CState,Peg1),
   btm_disk_on_peg(N,GState,Peg2),
   other_two_pegs(Peg1,Peg2,Peg3,Peg4),
   (Peg=Peg3;Peg=Peg4),
   arg(Peg,CState,CDisks),
   arg(Peg,GState,GDisks),
   PN is N-integer(sqrt(2*N)+0.5),
   Low is max(PN-2,1),
   Up is min(PN+2,N-1),
   between(Low,Up,Mid),
   (CDisks=[CBDisk|_],CBDisk=<Mid; CDisks=[]),
   (GDisks=[GBDisk|_],GBDisk=<Mid; GDisks=[]),
   Tower @= [I : I in Mid..(-1)..1],
   ItState=s(_,_,_,_),
   foreach(I in 1..4,
           (I==Peg->arg(I,ItState,Tower);
            arg(I,ItState,[])
           )
   ).
```

The arguments are similar to the ones in the `plan4` predicate, except `ItState` defines the intermediate state with the sub-tower, and `Peg` is the peg on which the sub-tower will be placed.

As mentioned above, there is an additional issue about where to place the sub-tower, based on the configurations. The program solves it in one of two ways. If the current configuration has a pre-existing sub-tower, a new one does not need to be created. Otherwise, the program non-deterministically tests both of the pegs that are currently serving as intermediate pegs for the optimal sub-tower location.

Given *n* disks, the program uses Rand's estimate of $n - \lfloor\sqrt{2n} + 0.5\rfloor$ for the sub-tower size. As discovered after the ASP Competition, this does not always generate the optimal solution. Therefore, the program was tested using different partition estimates. Table 2 shows the results[6].

Table 2: Partitions

Partition	Source	AVG Seconds	AVG Table Used (Bytes)	MAX Seconds	MAX Table Used (Bytes)	UNKNOWN
$N - \lfloor\sqrt{2 * N} + 0.5\rfloor$	[12]	0.0234	121135.4	0.047	207724	2
$N - \lceil\sqrt{2 * N + 0.25} - 0.5\rceil$	[20]	0.0209	121135.4	0.032	207724	2
$N - \lfloor\frac{\sqrt{8*N+1}-1}{2}\rfloor$	[1] [13] [22]	0.023	141369.4	0.032	206728	0
$N - K$, where K is the smallest integer such that $T_K \geq N$ (with tabling of results).	[23]	0.021566667	122005.1333	0.032	208868	2
$N - K$, where K is the largest integer such that $T_K \leq N$ (with tabling of results).	[1]	0.02285	142401.9333	0.032	207960	0
$N - K$, if $N = T_K$ for some K Otherwise, N - K, or N - (K + 1), where K is the largest integer such that $T_K < N$ (with tabling of results).	[17]	0.025833333	162988.3333	0.047	237360	0
Program Decides Without a Guide		0.039466667	4457510.067	0.124	16884172	0

For Table 2, two types of estimates were used. One type is a formula stated in terms of the number of disks. The other type relates the 4-peg Hanoi problem to the *triangular numbers*, which are of the form $T_k = \frac{k*(k+1)}{2}$ [13], by finding the triangular number that is closest to the number of disks. The estimates of the first type explicitly state the relation between the number of disks and the triangular numbers found by the second type [1]. In the final row of Table 2, the program was not given a guide for where to partition the disks. Instead, it tested every number between 1 and $n - 1$, where *n* is the number of disks.

There are a number of other guides for partitioning the disks [6] [9] [11] [17], but these were not tested. Some, such as [9], are too closely related to specific algorithms. Others, like [6], are too mathematically complex to test in terms of this program. Some of the remaining, like [11], are for specific cases. Some alternative partitions are clearly incorrect, as shown by Stockmeyer [17]. Instead of including Frame's and Stewart's estimates, this study incorpates Stockmeyer's estimate, which is based on those of Frame and Stewart [17].

The B-Prolog program allows for an error in the number, PN, returned by the estimates. It checks all values between $PN - 2$ and $PN + 2$ for an optimal partition number. If this error bound is removed, and the program uses the exact number that is found, or if the error bound is decreased to $PN - 1$ and $PN + 1$, then none of

[6] All of the tests described in this paper used B-Prolog version 7.5, which is the version used for the competition. Later versions of B-Prolog perform tabling in a different manner, decreasing the amounts of used table memory in some cases, and increasing the amounts used in other cases.

the functions in Table 2 solve all of the instances. It should be noted that three of the estimates in Table 2 fail for two of the instances, 17 and 22, even when the error is allowed.

The different approximations were tested for time and the amount of table memory that they used. On average, the explicit functions almost always used less memory than the partitions that calculated each triangular number until a certain condition was met. The maximum memory used was always smaller for the explicit functions than for the repeated calculations. One reason for this is that tabling was incorporated into the calculations of the triangular numbers in order to reduce the number of required calculations. As shown in Table 2, if the program tests every possible partition number, then it is guaranteed to find the optimal solution. However, the time and memory costs are much higher than they are if the program is given a partitioning guide.

Table 2 indicates that the best guide to use for the competition is the one originally provided by Rohl and Gedeon (as shown in row 3). It should be noted that, if floor and ceiling operations are removed, Liefvoort's solution (row 2) is mathematically equivalent to theirs. Since Rohl and Gedeon use the floor operation, while Liefvoort uses the ceiling operation, their solutions differ.

After the original program's error was removed, and the partitioning estimate was changed to that of Rohl and Gedeon, BPSolver's modified program was run on all 60 instances. Table 3 shows the results of these tests, as compared to Potassco's results. The results show that Rohl's and Gedeon's numbers are better than Rand's, because BPSolver now scores 100, instead of 94.

Table 3: Comparing B-Prolog's modified program with Potassco's program

Sys	AVG Seconds	MAX Seconds	Num Solved	SSolve	STime	Score
BPSolver	0.023	0.032	60	50	50	100
Potassco	10.61766667	102.867	60	50	37	87

4 Random Configurations and the Classic Problem

Each of the sixty instances generated for the 2011 ASP Competition consisted of configurations that would be created during the execution of the Frame-Stewart algorithm on the classic 4-peg problem. The configurations included pre-created sub-towers that the Frame-Stewart algorithm would have generated. This section will show that, although a certain partition number is shown to work on the ASP configurations, it is not guaranteed to solve every possible random configuration.

BPSolver's modified program was tested in two additional ways. It was run on randomly generated configurations, some of which might not be created during the execution of the Frame-Stewart algorithm, and it was run on the classic 4-peg problem.

4.1 Random Configurations

The first test was running the BPSolver program on randomly generated configurations. They were generated to match the ASP Competition's input files, which consisted of five sets of predicates [19]. The BPSolver program was tested on twenty random configurations. Since the configurations were random, some appeared to contain pre-existing intermediate sub-towers, while others did not. Two versions of BPSolver's program were tested. One version created the sub-towers using the formula originally posed by Rohl and Gedeon [13] with an error bound of two, and the other did not use any guide for where to partition the disks.

These tests had varied results. Only twelve of the twenty instances were solved by both programs. These instances had between 18 and 20 disks, while their solutions required between 120 and 257 steps. Another five instances were only solved by the program that did not use a partitioning guide. These instances had between 21 and 23 disks, and required between 387 and 408 steps. One instance, with 25 disks, was solved in 533 steps by the program that used a formula, while the program that did not use a guide ran out of memory. The remaining two instances, which had 24 and 25 disks, were solved by neither program.

The results appear to indicate that the 4-peg problem with random configurations requires a different approach than that of the Frame-Stewart algorithm. Due to the varying sizes of the disks on any peg in the starting configuration, it might not be easy to derive a mathematical formula for the creation of an intermediate sub-tower. If a guide is not used, it can be too computationally complex to test every possible set of moves. Disks may need to be repeatedly moved between the four pegs before a condition arises in which it is possible to create an intermediate sub-tower. Configurations such as those found in Fig. 2 might not be encountered during the regular execution of the Frame-Stewart algorithm. Therefore, unless modifications are introduced to the algorithm, the Frame-Stewart algorithm might not be the best to solve random configurations.

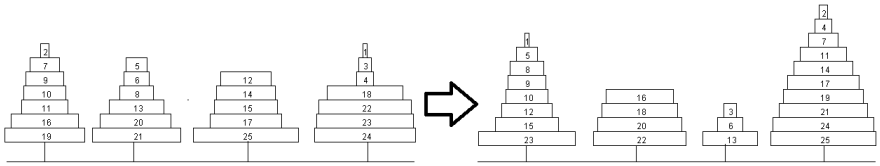


Fig. 2: Start and end configurations of an unsolved random instance

4.2 The Classic 4-peg Problem

The classic 4-peg problem was used to test the same two programs that were tested on the random configurations. For this problem, the programs were tested using an input of 30 disks. The program that used a formula was able to solve the problem using the

optimum possible number of moves, 1025 [7]. The one that had no guide ran out of memory.

Since the program that used Rohl's and Gedeon's guide was able to solve the 30-disk problem, it was tested on the classic problem using every number of disks between one and thirty. When these tests were run, the partitioning error bounds of +2 and -2 were removed. For each of the thirty instances, the program was able to solve the problem using the optimum possible number of moves, as proven by [7] and [8].

After these tests were performed, the classic 4-peg problem was used to test the same six partition estimates that were examined using the ASP problem in Table 2. Each estimate was tested on every number of disks between one and thirty. As opposed to the prior tests, these tests did not allow any error bound when calculating the partition number. Although three of the estimates failed to solve every single ASP Competition configuration, all six estimates generated optimal solutions for every instance of the classic problem. Korf has verified the minimum possible number of moves for any algorithm using inputs of up to 30 disks. By using the Frame-Stewart algorithm with any of the six partition estimates on the regular 4-peg Hanoi problem, a solution is obtained that is the optimal solution of any possible algorithm. Tabling and testing every partition number can be too computationally complex, but Rohl's and Gedeon's formula provides a good guide.

5 Alternative Approaches

Many authors have tried to solve the 4-peg and k-peg problems using their own algorithms. Although most of the approaches are not used in this paper, it is important to note their contributions. Following is a subset of the algorithms:

5.0.1 Rohl's and Gedeon's Algorithm. Rohl and Gedeon created recursive algorithms for both the 4-peg and the k-peg problems, using a form of Stewart's algorithm. This paper will focus on the algorithm for four pegs, wherein each recursive call builds a single sub-tower. If the current ordering of the pegs is (1, 2, 3, 4), the 4-peg algorithm is called to create a sub-tower with ordering (1, 4, 3, 2), the 3-peg algorithm is called on (1, 2, 3) to move the remaining pegs, and the 4-peg algorithm is called on (4, 3, 2, 1) to move the sub-tower to the destination peg [13]. This algorithm provided the partitioning guide utilized by BPSolver's modified program.

5.0.2 Lu's Algorithm. Lu produced an iterative approach for the 4-peg problem. Lu's algorithm shows a correlation between the disk moves and binary numbers. It inserts a number of logical fake disks at pre-determined locations in order to map the disk moves to the binary numbers [9].

5.0.3 Sarkar's Algorithm. Sarkar used a recursive algorithm for the k-peg problem that is similar to Frame's algorithm. The algorithm deterministically decides how to distribute the topmost disks over intermediate pegs, using summation functions, before distributing any disks. It then recursively calls itself to distribute the disks. Sarkar posed

the *serialization conjecture*, stating that an optimal algorithm will distribute disks onto pegs such that disks on each peg have consecutive sizes, and the size of the top disk on a peg is consecutive with the size of the bottom disk on the next peg. If this conjecture is valid, then Sarkar's algorithm is optimal [14]. Although this distribution method is efficient, this paper does not use it, because it does not easily extend to the configuration problem.

5.0.4 Wang's, Liu's, Yue's, Shao's, and Lu's Algorithm. Another iterative algorithm for the 4-peg puzzle involves arranging the disk numbers into an upper-triangular array. Based on this array, there is a "cross-correlation" between the solution of the problem represented by location (*i*, *j*) and the problems represented by locations (*i*, *j+1*), and (*i+1*, *j+1*). This correlation decreases the number of necessary calculations [22]. This is another efficient algorithm, although there is no clear way to extend it to the configuration problem.

6 Summary

For configurations directly based on the Frame-Stewart algorithm, Rohl's and Gedeon's partitioning estimate of $N - \lfloor \frac{\sqrt{8*N+1}-1}{2} \rfloor$ appears to be an optimal guide, even if it requires error bounds. Although all of the estimates in Table 2 generate optimal solutions for the regular 4-peg puzzle for up to 30 disks without using error bounds, their correctness does not seem to extend to the configuration problem. Perhaps the nature of the configuration problem changes the optimal partition location.

Testing BPSolver's program on random configurations further clarifies the issue. It appears that there are some configurations for which creating intermediate sub-towers may not be optimal. Unlike the problem as studied by Frame and Stewart, it is not clear where to place the intermediate sub-towers. The non-determinism involved can cause the computation problem to be too computationally complex. However, BPSolver's program with Rohl's and Gedeon's estimates runs quickly, and uses tabling to reduce the number of calculations. Therefore, it is a good guide for a dynamic programming solution for modified forms of the 4-peg Tower of Hanoi problem where non-determinism is required.

The B-Prolog program demonstrates the importance of tabling for declarative description of dynamic programming solutions. Like use of pattern databases in state-space search and conflict-driven clause learning and memoization in SAT solvers, tabling is a great technique for avoiding repeated exploration of the same states during search.

Acknowledgements

This research was supported in part by NSF (No.1018006)

References

1. Chu, I-Ping, and Richard Johnsonbaugh. The Four-Peg Tower of Hanoi Puzzle. *ACM SIGCSE Bulletin* Vol. 23, No. 3 (1991), 2-4.
2. Dunkel, Otto. Editorial Note Concerning Advanced Problem 3918. *The American Mathematical Monthly* Vol. 48, No. 3 (1941), 219.
3. Er, M. C. A Note on the Optimality of a Reve Algorithm. *The Computer Journal* Vol. 34, No. 6 (1991), 513.
4. Frame, J. S. Solution for Advanced Problem 3918. *The American Mathematical Monthly* Vol. 48, No. 3 (1941), 216-217.
5. Kaykobad, M, Rahman, S. T.-U., Bakhtiar, R.-A., and A. A. K. Majumdar. A Recursive Algorithm for the Multi-Peg Tower of Hanoi Problem. *International Journal of Computer Mathematics* Vol. 57, No. 1-2 (1995), 67-73.
6. Klavžar, Sandi, and Uroš Milutinović. Simple Explicit Formulas for the Frame-Stewart's Numbers. *Annals of Combinatorics* Vol. 6, No. 2 (2002), 157-167.
7. Korf, Richard E., and Ariel Felner. Recent Progress in Heuristic Search: A Case Study of the Four-Peg Towers of Hanoi Problem. *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (2007), 2324-2329.
8. Korf, Richard E., Zhang, Weixiong, Thayer, Ignacio, and Heath Hohwald. Frontier Search. *Journal of the ACM* Vol. 52, No. 5 (2005), 715-748.
9. Lu, Xue-Miao. An Iterative Solution for the 4-peg Towers of Hanoi. *The Computer Journal* Vol. 32, No. 2 (1989), 187-189.
10. Lunnon, W. F. The Reve's Puzzle. *The Computer Journal* Vol. 29, No. 5 (1986), 478.
11. Majumdar, A. A. K. Generalized Multi-peg Tower of Hanoi Problem. *Journal of the Australian Mathematical Society, Series B-Applied Mathematics* Vol. 38, No. 2 (1996), 201-208.
12. Rand, Michael. On the Frame-Stewart Algorithm for the Tower of Hanoi. Technical Report, Boston College (2009). Available via `https://www2.bc.edu/~grigsbyj/Rand_Final.pdf`.
13. Rohl, J. S., and T. D. Gedeon. The Reve's Puzzle. *The Computer Journal* Vol. 29, No. 2 (1986), 187-188.
14. Sarkar, U. K. On the Design of a Constructive Algorithm to Solve the Multi-peg Towers of Hanoi Problem. *Theoretical Computer Science* Vol. 237, No. 1-2 (2000), 407-421.
15. Stewart, B. M. Solution for Advanced Problem 3918. *The American Mathematical Monthly* Vol. 48, No. 3 (1941), 217-219.
16. Stockmeyer, P. K. The Tower of Hanoi: A Bibliography. Version 2.2 (2005). Available via `http://www.cs.wm.edu/~pkstoc/biblio2.pdf`.
17. Stockmeyer, Paul K. Variations on the Four-Post Tower of Hanoi Puzzle. *Proceedings of the 25th Southeastern International Conference on Combinatorics, Graph Theory, and Computing*. *Congressus Numerantium* Vol. 102 (1994), 3-12.
18. Third ASP Competition Detailed Scoring Regulations. ASP Competition 2011 Organizing Committee. Available via `https://www.mat.unical.it/aspcomp2011/files/scoringdetails.pdf`.
19. Truszczynski, Miroslaw, Smith, Shaden, and Alex Westlund. Tower of Hanoi: Problem Description. ASP Competition 2011. Available via `https://www.mat.unical.it/aspcomp2011/FinalProblemDescriptions/HanoiTower`.
20. van de Liefvoort, A. An Iterative Algorithm for the Reve's Puzzle. *The Computer Journal* Vol. 35, No. 1 (1992), 91-92.
21. van de Liefvoort, Appie. An Iterative Solution to the Four-Peg Tower of Hanoi Problem. *Proceedings of the 1990 ACM Annual Conference on Cooperation* (1990), 70-75.

22. Wang, Jun, Liu, Junpeng, Yue, Guoying, Shao, Liangshan, and Sukui Lu. A Non-recursive Algorithm for 4-Peg Hanoi Tower. *2007 International Conference on Intelligent Systems and Knowledge Engineering* (2007).
23. Zhang, Andrew. A Stratification of the Hanoi Graph for 4 Pegs. Technical Report, Columbia University (2008). Available via `http://www.math.columbia.edu/~jason/THreu.Zhang.pdf`.
24. Zhou, Neng-Fa. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming, Special Issue on Prolog Systems* Vol.12, No. 1-2 (2012), 189-218.
25. Zhou, Neng-Fa, Sato, Taisuke, and Yi-Dong Shen. Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming* Vol. 8, No. 1 (2008), 81-109.
26. Zhou, Neng-Fa, and Christian Theil Have. Efficient Tabling of Structured Data with Enhanced Hash-Consing. *ICLP*, 2012.
27. Zhou, Neng-Fa, Dovier, Agostino, and Yuanlin Zhang. BPSolver's Solutions to the Third ASP Competition Problems. *ALP Newsletter* (June 2011). Available via `http://www.cs.nmsu.edu/ALP/wp-content/uploads/2011/06/Bprolog.pdf`.
28. Zhou, Neng-Fa, Fruhman, Jonathan, and Ligon Liu. Program for Solving the 4-peg Tower of Hanoi Problem. ASP Competition 2011. Available via `http://www.probp.com/asp11/hanoi.pl`.

Extending the logical update view with transaction support

Jan Wielemaker

Web and Media group, VU University Amsterdam,
De Boelelaan 1081a,
1081 HV Amsterdam, The Netherlands,
J.Wielemaker@vu.nl

Abstract. Since the database update view was standardised in the Prolog ISO standard, the so called logical update view is available in all actively maintained Prolog systems. While this update view provided a well defined update semantics and allows for efficient handling of dynamic code, it does not help in maintaining consistency of the dynamic database. With the introduction of multiple threads and deployment of Prolog in continuously running server applications, consistency of the dynamic database becomes important.
In this article, we propose an extension to the generation-based implementation of the logical update view that supports transactions. Generation-based transactions have been implemented according to this description in the SWI-Prolog RDF store. The aim of this paper is to motivate transactions, outline an implementation and generate discussion on the desirable semantics and interface prior to implementation.

1 Introduction

Although Prolog can be considered a deductive database system, its practice with regard to database update semantics is rather poor. Old systems typically implemented the *immediate update view*, where changes to the clause-set become immediately visible to all goals on backtracking. This update view implies that a call to a dynamic predicate must leave a choice point to anticipate the possibility that a clause is added that matches the current goal. Quintus introduced[1] the notion of the *logical update view* [4], where the set of visible clauses for a goal is frozen at the start of the goal, which allows for pruning the choice-point on a dynamic predicate if no more clauses match the current goal. Through the ISO standard (section 7.5.4 of ISO/IEC 1321 l-l), the logical update view is adopted by all actively maintained implementations of the Prolog language.

The logical update view helps to realise efficient programs that depend on the dynamic database, but does not guarantee consistency of the database if multiple changes of the database are required to realise a transition from one consistent state to the next. A typical example is an application that realises a transfer between two accounts as shown below.

[1] http://dtai.cs.kuleuven.be/projects/ALP/newsletter/archive_93_96/net/systems/update.html

```
transfer(From, To, Amount) :-
        retract(balance(From, FromBalanceStart)),
        retract(balance(To, ToBalanceStart)),
        FromBalance is FromBalanceStart - Amount,
        ToBalance is ToBalanceStart + Amount,
        asserta(balance(From, FromBalance)),
        asserta(balance(To, ToBalance)).
```

Even in a single threaded environment, this code may be subject to timeouts, (resource-)exceptions and exceptions due to programming errors that result in an inconsistent database. Many Prolog programs contain a predicate to clean the dynamic database to avoid the need to restart the program during development. Still, such a predicate needs to be kept consistent with the used set of dynamic predicates and provides no easy solution if the database is not empty at the start.

Obviously, in concurrent applications we need additional measures to guarantee consistency with concurrent transfer requests and enquiries for the current balance. Below we give a possible solution based on mutexes. Another solution is to introduce a `bank` thread and realise transfer as well as enquiries with message passing to the `bank` thread.

```
mt_transfer(From, To, Amount) :-
        with_mutex(bank, transfer(From, To, Amount)).

mt_balance(Account, Balance) :-
        with_mutex(bank, balance(Account, Balance)).
```

As we need to serialise update operations in a multi threaded context, update operations that require significant time to complete seriously harm concurrency.

With transactions, we can rewrite the above using the code below, where we do not need any precautions for reading the current balance.[2] Consistency is guaranteed because, as we will explain later, two transactions that retract the same clause are considered to conflict.

```
mt_transfer(From, To, Amount) :-
        transaction(transfer(From, To, Amount), true, [restart(true)]).
```

In the remainder of this article, we first describe related work. Next, we define the desired semantics of transactions in Prolog, followed by a description how these semantics can be realised using generations. In section 5, we propose a concrete set of predicates to make transactions available to the Prolog programmer. We conclude with implementation experience in the SWI-Prolog RDF store and a discussion section.

[2] Reading multiple values from a single consistent view still requires a transaction. See section 3.1.

2 Related work

The most comprehensive overview of transactions in relation to Prolog we found is [2], which introduces "Transaction logic". Transaction logic has been implemented as a prototype for XSB Prolog [3]. This is a much more fundamental solution in dealing with update semantics than what we propose in this article. In [2], we also find descriptions of related work, notably *Dynamic Prolog* and an extension to Datalog by Naqvi and Krishnamurthy. These systems too introduce additional logic and are not targeted to deal with concurrency.

Contrary to these systems, we propose something that is easy to implement on engines that already provide the logical update view and is easy to understand and use for a typical Prolog programmer. What we do learn from these systems is that a backtrackable dynamic database has promising applications. We not propose to support backtracking modifications to the dynamic database yet, but our proposal simplifies later implementation thereof, while the transaction interface may provide an adequate way to scope backtracking. See **transaction/3** described in section 5.

3 Transaction semantics

Commonly seen properties of transactions are known by the term ACID,[3] summarised below. We want to realise all these properties, except for *Durability*.

Atomic Either all modifications in the transaction persist or none of the modifications.

Consistency A successful transaction brings the system from one consistent state into the next.

Isolation The modifications made inside a transaction are not visible to the outside world before the transaction is committed. Concurrent access always sees a consistent database.

Durability The effects of a committed transaction remain permanently visible.

In addition, code that is executed in the context of a transaction should behave according to the traditional Prolog (logical view) update semantics and it must be possible to nest transactions, such that code that creates a transaction can be called from any context, both outside and inside a transaction.

An obvious baseline interface for dealing with transactions is to introduce a meta-predicate `transaction(`*:Goal*). If *Goal* succeeds, the transaction is committed. If *Goal* fails or throws an exception, the transaction is rolled back and **transaction/1** fails or re-throws the exception. In our view, **transaction/1** is logically equivalent to **once/1** (i.e., it prunes possibly remaining choice points) because database actions are still considered side-effects. Too much real-world code is intended to be deterministic, but leaves unwanted choice points. This is also the reason why **with_mutex/2** prunes choice points.

Note that it is easy to see useful application scenarios for non-deterministic transactions. For example, generate-and-test applications that use the database could be implemented using the skeleton code below. To support this style of programming, failing

[3] `http://en.wikipedia.org/wiki/ACID`

into a transaction should atomically make the changes invisible to the outside and all changes after the last choice point inside the transaction must be discarded. This can be implemented by extending each choice point with a reference into the change-log maintained by the current transaction.

```
generate_and_test :-
        transaction(generate_world),
        satisfying_world,
        !.
```

Given our proposed once-based semantics, we can still improve considerably on this use-case compared to traditional Prolog using a side-effect free generator. This results in the following skeleton:

```
generate_and_test :-
        generate_world(World),
        transaction(( assert_world(World),
                      satisfying_world
                    )),
        !.
```

3.1 Snapshots

We can exploit the isolation feature of transactions to realise *snapshots*. A predicate `snapshot(`*:Goal*) executes *Goal* as **once/1** without globally visible affects on the dynamic database. This feature is a supplement to SWI-Prolog's *thread local* predicates, predicates that have a different set of clauses in each thread. Snapshots provide a comfortable primitive for computations that make temporary use of the dynamic database. At the same time they make such code thread-safe as well as safe for failed or incomplete cleanup due to exceptions or programming errors.

Snapshots also form a natural abstraction to read multiple values from a consistent state of the dynamic database. For example, the summed balance of a list of accounts can be computed using the code below. The snapshot isolation guarantees that the result correctly represents that summed balance at the time that the snapshot was started.

```
summed_balance(Accounts, Sum) :-
        snapshot(maplist(balance, Accounts, Balances)),
        sum_list(Balances, Sum).
```

Note that this sum may be outdated before the isolated goal finishes. Still, it represents a figure that was true at a particular point in time, while unprotected execution can compute a value that was never correct. For example, consider the sequence of events below, where to summed balance is 10$ too high.

1. The 'summer' fetches the balance of A
2. A concurrent operation transfers 10$ from A to B
3. The 'summer' fetches the balance of B

4 Generation based transactions

A common technique used to implement the Prolog logical update view is to tag each clause with two integers: the generation in which it was born and the generation in which it died. A new goal saves the current generation and only considers clauses created before and not died before its generation. This is clearly described in [1]. Below, we outline the steps to add transactions to this picture.

First, we split the generation range into two areas: the low values, 0..G_TBASE (transaction base generation) are used for globally visible clauses. Generations above G_TBASE are used for generations, where we split the space further by thread-id. E.g., the generation for the 10th modification inside a transaction executed by thread 3 is G_TBASE+3*G_TMAX+10.

Isolation Isolated behaviour inside a transaction is achieved by setting the modification generation to the next thread write generation. Code operating outside a transaction does not see these modifications because they are time-stamped 'in the future'. Code operating inside the transaction combines the global view at the start of the transaction with changes made inside the transaction, i.e., changes in the range G_TBASE+⟨*tid*⟩*G_TMAX..G_TBASE+(⟨*tid*⟩+1)*G_TMAX.

Atomic Committing a transaction renumbers all modifications to the current global write generation and then increments the generation. This implies that commit operations must be serialised (locked). All modifications become atomically visible at the moment that the global generation is incremented. If a transaction is discarded, all asserted clauses are made available for garbage collection and the generation of all retracted clauses is reset to infinity.

Consistency The above does not provide consistency guarantees. We add a global consistency check by disallowing multiple retracts of the same clause. This implies that an attempt to retract an already retracted clause inside a transaction or while the transaction commits causes the transaction to be aborted. This constraint ensures that code that *updates* the database by retracting a value, computing the new value and asserting this becomes safe. For example, this deals efficiently with global counters or the balance example from section 1. Note that disallowing multiple retracts is also needed because there is only one placeholder to store the 'died generation'. Similar to relational databases, we can add an integrity constraint, introducing `transaction`(*:Goal, :Constraint*), where *Constraint* is executed while the global commit lock is held.

Nesting Where we need distinct generation ranges for concurrently executing transactions, we can use the generation range of the parent transaction for a nested transaction because execution as **once/1** guarantees strict nesting. Nested transactions merely need to remember where the nested transaction started. Committing is a no-op, while discarding is the same as discarding an outer transaction, but only affecting modifications after the start of the nested transaction.

Implication for visibility rules The logic to decide that a clause is visible does not change for queries outside transactions because manipulations inside transactions are 'in the future'. Inside a transaction, we must exclude globally visible clauses that have died inside the transaction (i.e., between the transaction start generation and the current generation) and include clauses that are created and not yet retracted in the transaction.

5 Proposed predicates

We propose to add the following three predicates to Prolog. In the description below, predicate arguments are prefixed with a *mode annotation*. The `:` annotation means that the argument is module-sensitive, e.g., *:Goal* means that *Goal* is called in the module that calls the transaction interface predicate. The modes `+`, `-` and `?` specifies that the argument is 'input', 'output' and 'either input or output'.

transaction(*:Goal*)
: Execute *Goal* in a transaction. *Goal* is executed as by **once/1**. Changes to the dynamic database become visible atomically when *Goal* succeeds. Changes are discarded if *Goal* does not succeed.

transaction(*:Goal, :Constraint*)
: Run *Goal* as **transaction/1**. If *Goal* succeeds, execute *Constraint* while holding the `transaction` mutex (see **with_mutex/2**[4]). If *Constraint* succeeds, the transaction is committed. Otherwise, the transaction is discarded. If *Constraint* fails, throw the error `error(transaction_error(constraint, failed), _)`. If *Constraint* throws an exception, rethrow this exception.

transaction(*:Goal, :Constraint, +Options*)
: As **transaction/3**, processing the following options:

  restart(*+Boolean*)
  : If `true`, catch errors that unify with `error(transaction_error(_,_), _)` and restart the transaction.

  id(*Term*)
  : Give the transaction an identifier. This identifier is made available through **transaction_property/2**. There are no restrictions on the type or instantiation of *Term*.

snapshot(*:Goal*)
: Execute *Goal* as **once/1**, isolating changes to the dynamic database and discarding these changes when *Goal* completes, regardless how.

transaction_property(*?Transaction, ?Property*)
: True when this goal is executing inside a transaction identified by the opaque ground term *Transaction* and has given *Property*. Defined properties are:

  level(*-Level*)
  : Transaction is nested at this level. The outermost transaction has level 1. This property is always present.

  modified(*-Boolean*)
  : True if the transaction has modified the dynamic database.

[4] `http://www.swi-prolog.org/pldoc/doc_for?object=with_mutex/2`

modifications(*-List*)
List expresses all modifications executed inside the transaction. Each element is either a term `retract`(*Term*),`asserta`(*Term*) or `assertz`(*Term*). Clauses that are both asserted and erased inside the transaction are omitted.

id(*?Id*)
Transaction has been given the current *Id* using **transaction/3**.

If any of these predicate encounters a conflicting retract operation, the exception `error(transaction_error(conflict, PredicateIndicator), _)` is generated.

6 Implementation results: the SWI-Prolog RDF-DB

The SWI-Prolog RDF database [5] is a dedicated C-based implementation of a single dynamic predicate `rdf`(*?Subject, ?Predicate, ?Object*). The dedicated implementation was introduced to reduce memory usage and improve performance by exploiting known features of this predicate. For example, all arguments are ground, and *Subject* and *Predicate* are know to be atoms. Also, all 'clauses' are unit clauses (i.e., there are no rules). Quite early in the development of the RDF store we added transactions to provide better consistency and grouping of modifications. These features were crucial for robustness and 'undo' support in the graphical triple editor Triple20 [6]. Initially, the RDF store did not support concurrency. Later, this was added based on 'read/write locks', i.e., multiple readers or one writer may access the database at any point in time.

With version 3[5] of the RDF store, developed last year, we realised the logical update view also for the external **rdf/3** predicate and we implemented transactions following this article. In addition, we realised concurrent garbage collection of dead triples. The garbage collector examines the running queries and transactions to find the oldest active generation and walks the linked lists of the index hash-tables, removing dead triples from these lists. Actual reclaiming of the dead triples is left to the Boehm-Demers-Weiser conservative garbage collector.[6]

7 Discussion

We have described an extension to the generation-based logical update view available in todays Prolog system that realises transactions. There is no additional memory usage needed for clauses. The engine (each engine in multi threaded Prolog systems) is required to maintain a stack of transaction records, where each transaction remembers the global generation in which is was created and set of affected clauses (either asserted or retracted). The visibility test of a clause for goals outside transactions is equal to the test required for realising the logical update view and requires an additional test of the same cost if a transaction is in progress.

[5] Available from `http://www.swi-prolog.org/git/packages/semweb.git`, branch `version3`.

[6] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

The described implementation of transactions realises ACI of the ACID model (atomic, consistency and isolation, but not durability). Durability can be realised by using a constraint goal that uses **transaction_property/2** to examine the modifications and write the modifications to a journal file or external persistent store.

Our implementation has two limitations: (1) goals in a transaction are executed as **once/1** (pruning choice-points) and (2) it is not possible for multiple transactions to retract the same clause. Supporting non-deterministic transactions requires additional changes to Prolog choice points, but transactions already maintain a list of modifications to realise commit and rollback and **transaction/3** already provides an extensible interface to activate this behaviour. Supporting multiple retracts is possible by using a list to represent the 'died' generations of the clause. This is hardly useful for transactions because multiple concurrent retracts indicate a conflicting update. However, this limitation is a serious restriction for snapshots (section 3.1).

We have implemented this transaction system for the SWI-Prolog RDF store, where it functions as expected. We believe that transactions will greatly simplify the implementation of concurrent programs that use the dynamic database as a shared store. At the same time it eliminates the need for serialisation of code, improving concurrent performance. In our experience with Triple20, transactions are useful in single threaded applications to maintain consistency of the database. Notably, consistency of the dynamic storage is maintained when an edit operation fails due to a programming error or an abort initiated from the debugger. This allows for fixing the problem and retrying the operation without restarting the application.

We plan to implement the outlined features in SWI-Prolog in the near future.

Acknowledgements

This research was partly performed in the context of the COMBINE project supported by the ONR Global NICOP grant N62909-11-1-7060. This publication was supported by the Dutch national program COMMIT.

I would like to thank Jacco van Ossenbruggen, Michiel Hildebrand and Willem van Hage for their feedback in redesigning the transaction support for the SWI-Prolog RDF store.

References

1. Egon Boerger and Bart Demoen. A framework to specify database update views for prolog. In Jan Maluszynski and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 147–158. Springer Berlin / Heidelberg, 1991. 10.1007/3-540-54444-5_95.
2. Anthony J. Bonner, Michael Kifer, and Mariano Consens. Database programming in transaction logic. In *In Proc. 4th Int. Workshop on Database Programming Languages*, pages 309–337, 1993.
3. Samuel Y.K. Hung. Implementation and performance of transaction logic in prolog. Master's thesis, Department of Computer Science, University of Toronto, 1996.
4. Timothy G. Lindholm and Richard A. O'Keefe. Efficient implementation of a defensible semantics for dynamic prolog code. In *ICLP*, pages 21–39, 1987.

5. Jan Wielemaker, Guus Schreiber, and Bob J. Wielinga. Prolog-based infrastructure for rdf: Scalability and performance. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 644–658. Springer, 2003.
6. Jan Wielemaker, Guus Schreiber, and Bob J. Wielinga. Using triples for implementation: The triple20 ontology-manipulation tool. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 773–785. Springer, 2005.

Efficient Partial Order CDCL Using Assertion Level Choice Heuristics*

Anthony Monnet and Roger Villemaire

Université du Québec á Montréal, Montreal, Canada
anthonymonnet@aol.fr
villemaire.roger@uqam.ca

Abstract. We previously designed Partial Order Conflict Driven Clause Learning (PO-CDCL), a variation of the satisfiability solving CDCL algorithm with a partial order on decision levels, and showed that it can speed up the solving on problems with a high independence between decision levels. In this paper, we more thoroughly analyze the reasons of the efficiency of PO-CDCL. Of particular importance is that the partial order introduces several candidates for the assertion level. By evaluating different heuristics for this choice, we show that the assertion level selection has an important impact on solving and that a carefully designed heuristic can significantly improve performances on relevant benchmarks.

1 Introduction

The SAT problem consists in deciding if a given propositional formula expressed in conjunctive normal form is satisfiable, i.e. if there exists a truth assignment that makes the formula true. Furthermore, a satisfying assignment, or model, has to be returned if the formula is satisfiable. Many decision problems can be encoded using a propositional formula, such that this formula is satisfiable iff the considered problem has a solution.

Conflict-driven clause learning (CDCL) [9] is the algorithm used by the most efficient complete SAT solvers. Unlike basic depth-first search that only undoes the last decision when a conflict is reached, CDCL is able to analyze the reasons for this conflict and to define the assertion level, which is the second to last decision level involved in this conflict. It then backtracks directly to this assertion level, often undoing several decision levels at once, in order to ensure that this conflict will not be encountered again in this branch of the search. It therefore performs a much more efficient pruning of the search space than regular depth-first search, often leading to a significantly faster solving of problems.

CDCL has however the negative side-effect of destroying parts of the current partial assignment not directly related to the conflict. Indeed, by returning straight to the assertion level, it entirely destroys all instantiations in subsequent decision levels. By definition, none of them were directly involved in the conflict, except for the decision level where the conflict was discovered. In the worst case, these instantiations may even

* We gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada on this research.

belong to a different connected component of the problem and couldn't possibly be affected by the conflict resolution, even indirectly. CDCL thus can cause the unnecessary deletion of previous parts of the search, which may prevent the detection of some conflicts or the completion of a satisfying assignment, ultimately slowing down the solving process.

This deletion of unrelated parts of the search is caused by the implicit total ordering on successive decisions during the search, and this total order can be relaxed without damaging the correctness, completeness and termination of the algorithm. We therefore designed partial order CDCL (PO-CDCL) [11], a variant of CDCL maintaining a partial order between decision levels, which allows to locally undo less instantiations during a conflict-directed backtrack. In practice, some SAT problems (for instance encodings from the formal verification of superscalar microprocessors [17]) have a relatively sparse dependency between decision levels during solving, and we showed that PO-CDCL significantly decreases the solving time on these instances.

The aim of this paper is twofold. First, we show that the efficiency of PO-CDCL is due to the fact that it dramatically reduces the search efforts needed to reach successive conflicts and hence prune the search space. Secondly, we consider a new parameter of the algorithm introduced by the partial order: unlike in a regular CDCL, the assertion level of a conflict clause is not uniquely defined and can be chosen using various heuristics. We show that this choice has a significant impact on the search, and that heuristics affecting the average amount of instantiations undone by conflicts can further significantly improve the performance of PO-CDCL. Interestingly, the solving of satisfiable problems is improved when this average amount of undone instantiations increases, while unsatisfiability is proved faster when it decreases. Moreover, this quantity is most efficiently controlled indirectly by choosing assertion levels that maximize or minimize the number of additional dependencies they would introduce between decision levels.

The remainder of this paper is organized as follows: Section 2 introduces the PO-CDCL algorithm. Section 3 reviews related methods seeking to reduce the amount of instantiations deleted during a non-chronological backtracking algorithm in CSP and SAT. Finally, section 4 presents experimental results obtained with the implementation of PO-CDCL in a state-of-the-art CDCL solver using various heuristics for the choice of the assertion level. These results are used to analyze the causes of the efficiency of PO-CDCL with various assertion level heuristics on satisfiable and unsatisfiable SAT instances with low level interdependencies.

2 PO-CDCL

CDCL [9] is a satisfiability solving algorithm based on the older depth-first search DPLL [3], enhanced with conflict-directed backtracking and clause learning. It successively assigns arbitrary values to variables (it takes decisions) until either a clause is violated or all variables are assigned. After each decision, an exhaustive round of unit propagation is performed to deduce all possible consequences of the current assignment using this inference rule. A decision level is the set formed by a decision and all the propagations it entails.

Algorithm 1 PO-CDCL

```
 1: σ ← ∅ { begin with the empty assignment }
 2: λ = 0 { λ is the current decision level }
 3: loop
 4:    c ←PROPAGATE /* propagate new instantiations */
 5:    if c ≠ NIL then { a conflict was found during propagations }
 6:       if λ = 0 then { conflict at decision level 0 }
 7:          return false { unsatisfiable problem }
 8:       else
 9:          γ ← ANALYZE(c) { infer the conflict clause γ }
10:          candidates ← all Δ-maximal elements in levels(γ) \ {λ}
11:          choose a in candidates { a is the assertion level }
12:          for l >_Δ a do
13:             delete level l
14:          λ ← a { a becomes the current level }
15:          LEARN(γ)
16:          PROPAGATEASSERTION(γ)
17:    else { no conflict during propagations }
18:       if all variables are instantiated then
19:          return σ { σ is a model }
20:       else
21:          λ ← NEWLEVEL
22:          DECIDE(λ)
```

Algorithm 2 PROPAGATE

```
 1: Π ←{instantiations not yet propagated}
 2: while {Π ≠ ∅} do
 3:    choose l ∈ Π
 4:    for all clauses c s.t. ¬l is watched in c do
 5:       w ← the second watched literal in c
 6:       if σ(w) = true then
 7:          set level(w) <_Δ λ
 8:       else
 9:          Ω ← {l' ∈ c | σ(l') ≠ false} \ {w}
10:          { Ω is the set of literals that could replace ¬l }
11:          if Ω = ∅ then { no other literal in c can be watched }
12:             if σ(w) = undef then { c is unit }
13:                σ(w) ← true { w is propagated by c }
14:                for l ∈ levels(c) \ {λ} do
15:                   set l <_Δ λ
16:                Π ← Π ∪ {w}
17:             else
18:                return c { c is a conflict }
19:          else
20:             choose w'∈Ω
21:             ω(c) ← {w, w'} { w' is watched instead of ¬l }
22:    Π ← Π \ {l}
23: return NIL { no conflict occured }
```

Unit clauses are efficiently detected using watched literals [13], a method keeping track of two not instantiated literals in each clause that isn't already satisfied. When a literal l is instantiated, a clause c cannot become unit unless it contains the opposite literal $\neg l$ and this literal is watched in c. The algorithm thus only has to check clauses containing $\neg l$ as a watched literal for possible unit propagations.

A conflict occurs when all literals in a clause are false. CDCL then infers a conflict clause γ, which is a logical consequence of the original formula, is also false under the current assignment and has only one literal instantiated at the current decision level. The second largest decision level represented in γ is called the assertion level. The conflict is resolved by undoing all decision levels above the assertion level. γ becomes unit, it is propagated and the search continues at the assertion level. The algorithm terminates either when all variables are assigned without causing any conflict (the formula is satisfied by this assignment) or when a conflict occurs at decision level 0 (the formula is unsatisfiable).

The pseudocode of PO-CDCL is given in Alg. 1. It consists in a few modifications of the regular CDCL algorithm. A partial order Δ keeps track of dependencies between decision levels and is used to determine the assertion level and the levels to delete during conflicts. Dependencies are added during the unit propagation phase detailed in Alg. 2. A level i depends on a level j (noted $j <_\Delta i$) when level j had an influence on unit propagations at level i. This can happen in two cases.

First, when a variable l is propagated by a unit clause c, this propagation obviously depends of all other literals in c. For all literals $l' \in c \setminus \{l\}$ whose decision level is different from the current decision level λ, the dependency $level(l') <_\Delta \lambda$ is added to Δ. This case is handled by lines 14 and 15 of Alg. 2.

Secondly, when a false watched literal $\neg l$ at level λ doesn't need to be replaced in a clause c because w, the second watched literal in c, is true, the dependency $level(w) <_\Delta \lambda$ is added (line 7 of Alg. 2). Intuitively, this dependency means that the true watched literal w avoided a watched literal replacement at level λ and therefore had an impact on the unit propagations at this level. More technically, this dependency ensures that the clause will remain correctly watched by forbidding to uninstantiate w while keeping $\neg l$ instantiated.

With a partial order on decision levels, the backtrack phase only requires to delete levels that depend on the assertion level (and of course the conflict level itself). This deletion (at lines 12 and 13 of Alg. 1) is necessary to keep the consistency of the algorithm by preventing circular dependencies between levels.

Finally, the last modification affects the definition of the assertion level. This level has to be involved in the conflict clause, and the conflict clause must become unit after the backtrack. This implies that no decision level occuring in the conflict clause must be undone by the backtrack, except for the conflict level λ. In a total order CDCL, the assertion level is uniquely defined as the second largest decision level in the conflict clause. With a partial order, however, any decision level in the conflict clause can be chosen as the assertion level, provided that no other level involved in the conflict, except λ, depends on it. In other words, the assertion level can be any maximal element of $<_\Delta$ restricted to the set of conflict clause levels different from λ (lines 10 and 11 of Alg. 1).

Similarly to the original CDCL algorithm, PO-CDCL is complete, correct and always terminates [11].

3 Related Works

PO-CDCL is conceptually related to some variations of the Conflict-Direct Backjumping (CBJ) algorithm for CSP solving which, similarly to CDCL for SAT, resolves conflicts by computing a nogood (equivalent of the conflict clause) and deleting the entire search progress starting at the culprit variable decision (roughly equivalent of the decision at the conflict level). In the case of CSPs, search progress consists not only of variable assignments, but also of values eliminated from domains of variables.

Dynamic Backtracking (DB) [5], in contrast with CBJ, only undoes the culprit variable and restores only eliminated values for which the culprit variable was part of the nogood. This strategy is equivalent to dynamically moving the culprit variable to the end of the search branch before undoing it, provided a limited amount of search information is deleted. It has the advantage of only partially undoing the work made after the culprit variable. Similarly to PO-CDCL, it minimizes the quantity of undone search progress by relaxing the strict total order on variable decisions. The main difference is that DB is defined as a search-only algorithm without any inference; therefore the conflict can always be resolved without undoing any other decision than the culprit variable. Also note that the usual total order is considered during the analysis phase; unlike the assertion level in PO-CDCL, the culprit variable in DB remains thus uniquely defined.

Partial Order Backtracking (POB) [10] similarly only uninstantiates the culprit variable for each conflict and only restores values whose elimination depended on it. The difference is that it initially allows to choose the culprit variable amongst all variables in the nogood, but progressively sets precedence constraints between variables in order to ensure termination. This freedom in the choice of the culprit variable is stronger than the freedom PO-CDCL offers for choosing the assertion level. It however however comes with a strong permanent and increasing constraint on decision heuristics, whereas constraints set by PO-CDCL between decision levels only apply until these decision levels are undone, and hence have no impact on the choice of decision variables.

Tree decompositions methods integrated within CDCL [6,8,4,12] and CBJ [7] solvers also indirectly limit the quantity of unrelated instantiations undone during a backtrack. Decompositions [16] are used to compute recursive separators of the instance, i.e. sets of variables whose instantiation breaks the problem in several connected components. These methods start the search by instantiating all separator variables, and then completely instantiate a connected component before making any decision in another component. When a conflict occurs in a connected component, the resulting backtrack then can't destroy any part of the search in other components thanks to this constrained ordering. Besides scalability issues which make it very difficult to efficiently compute useful decompositions on large SAT problems [12], tree decompositions only capture the static connectivity of a problem and therefore can't take into account the polarity of instantiations and the many propagations they cause. At any point of the search, the actual connectivity is likely to be much more sparse than predicted by decompositions. Therefore, a conflict in a connected component may actually delete instantiations

in another component. PO-CDCL, on the other hand, considers the exact connectivity at any time of the search. It also distinguishes sets of variables that haven't interacted yet in the current search branch even if they belong to the same connected component; it considers actual interactions between already instantiated variables rather than potential interactions between still unassigned variables.

Finally, phase saving [15], in contrast with tree decompositions, is a very lightweight approach. It simply memorizes the last polarity assigned to a variable and reuses it if the variable is picked for a decision. Phase saving actually doesn't prevent instantiations from being undone, but makes it possible to rediscover the deleted instantiations later. It thus allows to recover search progress that was lost during a conflict resolution. However, unlike partial order CDCL, this recovery doesn't save the computational cost of repeating the time-consuming propagation phase. Also, phase saving memorizes the polarity of all variables, even if they were actually involved in the conflict. This side effect sometimes decreases solving performance, as reported by the authors [15].

Note that, at the opposite, some strategies have been designed to enhance SAT solving by increasing the quantity of instantiations undone during conflict-directed backtracks [14,2].

4 PO-CDCL Analysis and Assertion Level Heuristics

The PO-CDCL algorithm was implemented by introducing a partial order on decision levels in the state-of-the-art CDCL solver GLUCOSE 1.0 [1]. The resulting PO-CDCL solver is named PO-GLUCOSE and its source code is available at `http://www.info2.uqam.ca/~villemaire_r/Recherche/SAT/120619generalized_glucose.tar.gz`. In this implementation, level dependencies are stored in three structures: two directed adjacency lists, representing the relation in both directions, and one boolean matrix. The combination of these structures allows to perform efficiently all operations on the partial relation: somes cases require to check the relation between a precise pair of decision levels, which can be done in constant time using the matrix. At the opposite, the algorithm sometimes requires to list of all levels depending on a given level, in which case using the adjacency list is obviously more efficiently, particularly when there are many active decision levels with little interdependence. Note that only direct dependencies are stored; transitive dependencies are only needed during conflict resolution on a small subset of variables and it is much more efficient to compute this partial transitive closure when it is required than to enforce and store transitivity during the entire algorithm.

As the size of the matrix grows quadratically with the number of decision levels, our implementation disables it if this number reaches a predefinite threshold. The algorithm then proceeds using only adjacency lists, which is slightly less efficient but significantly better than exhausting primary memory. Theoretically, the size of adjacency lists could also grow quadratically in the case of dense dependencies between decision levels; however, it seems that in practice the number of decision levels tends to decrease when this density grows. The memory requirement of adjacency lists thus remains relatively moderate.

The remaining of this section presents and compares experimental results obtained with this implementation and with the original GLUCOSE solver. We will more particularly focus on the impact of assertion level choice heuristics on the overall behaviour and performance of the algorithm. All tests were run on a 3.16 GHz Intel Core 2 Duo CPU with 3 GB of RAM, running a Ubuntu 11.10 OS, with a time limit of 1 hour for each execution (not including the preprocessing phase, which is identical for all tested variants).

We previously noticed [11] that since PO-CDCL is designed to take advantage of the independence between decision levels during solving, it performs best on problems where this independence is relatively high. If we consider the partial order Δ as a set of ordered pairs of decision levels, such that the first level in each pair depends of the second level of the pair, the cardinality of Δ can be used as a measure of this independence. Benchmarks from formal verification of superscalar microprocessors [17] are an example of problems with a very sparse relationship between levels, possibly because of the high parallelism in verified models. Therefore, the following experiments were conducted on 6 series of these benchmarks:

- pipe_unsat_1.0 and pipe_unsat_1.1 verify correct specifications of various-sized superscalar microprocessors with two different encoding variants;
- pipe_sat_1.0 and pipe_sat_1.1 represent ten different buggy variants of the size 12 case, again encoded in two different ways;
- pipe_ooo_unsat_1.0 and pipe_ooo_unsat_1.1 are two different encodings verifying the correctness of various-sized superscalar microprocessors handling out-of-order execution of instructions.

Benchmarks verifying correct and buggy specifications are respectively unsatisfiable and satisfiable.

GLUCOSE implements the phase saving strategy mentioned in section 3. We disabled phase saving in PO-GLUCOSE because partial order CDCL was partly designed as an alternative to phase saving. Moreover, preliminary tests indicated that PO-GLUCOSE often performs significantly better with phase saving disabled. To make sure the performance differences we observe are not simply caused by the presence or abscence of phase saving rather than by the partial order, we compared PO-GLUCOSE with the original GLUCOSE, but also with a variant in which phase saving is disabled.

In PO-GLUCOSE, the partial order management causes a significant calculation overhead during solving. Indeed, each propagation requires to check and possibly add several level dependencies. As a result, given the same execution time on the same instance, PO-Glucose performs on average about 40% less clause checks (i.e. the number of executions of the innermost **for** loop at lines 4 to 21 of Alg. 2) than GLUCOSE. We think this overhead can't be significantly reduced unless we find some lazy strategy to manage dependencies. Therefore, besides the CPU time used to solve each instance, we also report the number of clauses checked for possible propagations during solving. This quantity gives some insight about which proportion of the PO-GLUCOSE solving time is spent in the search itself and to what extent this time is due to dependency management.

family	#inst	TO		TO-phase		PO		PO-least-undos		PO-most-undos		PO-least-deps		PO-most-deps	
		#to	time	#to	time	#to	time	#to	time	#to	time	#to	time	#to	time
pipe_sat_1.0	10	6	25 364	0	6 887	0	6 601	0	7 334	0	2 042	0	1 264	2	10 399
pipe_sat_1.1	10	1	7 258	0	1 182	1	3 766	1	4 010	0	186	0	185	1	3 820
pipe_unsat_1.0	13	5	23 172	7	25 627	5	19 456	5	19 697	5	19 192	5	20 338	4	17 742
pipe_unsat_1.1	14	5	20 706	7	28 460	6	23 591	6	23 130	6	22 837	6	23 149	6	22 198
pipe_ooo_unsat_1.0	9	2	10 757	1	6 989	2	11 670	3	13 321	2	10 613	2	10 799	2	10 420
pipe_ooo_unsat_1.1	10	1	11 457	4	30 563	1	12 153	2	16 185	2	15 909	2	16 485	1	11 592
total	66	20	98 714	19	99 708	15	77 237	17	83 677	15	70 870	15	72 236	16	76 196

Table 1: Compared performances of GLUCOSE without (*TO*) and with (*TO-phase*) phase saving, PO-GLUCOSE with the default chronological assertion level heuristic (*PO*) and with 4 other heuristics based on the amount of instantiations undone by the backtrack (*PO-least-undos*, *PO-most-undos*) or on the number of level dependencies added (*PO-least-deps*, *PO-most-deps*). For each *series* of benchmarks, containing *#inst* instances, the number of timeouts (*#to*) and the total solving time in seconds (*time*) is given.

series	#inst	TO		TO-phase		PO		PO-least-undos		PO-most-undos		PO-least-deps		PO-most-deps	
		#to	checks	#to	checks	#to	checks	#to	checks	#to	checks	#to	checks	#to	checks
pipe_sat_1.0	10	6	60 962	0	174 962	0	74 034	0	104 876	0	23 970	0	12 816	2	45 065
pipe_sat_1.1	10	1	257 229	0	38 492	1	1 438	1	4 542	0	1 361	0	1 123	1	1 938
pipe_unsat_1.0	13	5	204 171	7	336 799	5	34 804	5	39 208	5	23 256	5	52 394	4	58 161
pipe_unsat_1.1	14	5	95 137	7	384 249	6	51 028	6	35 829	6	26 370	6	34 717	6	11 874
pipe_ooo_unsat_1.0	9	2	124 488	1	107 063	2	75 783	3	48 229	2	55 363	2	56 183	2	51 501
pipe_ooo_unsat_1.1	10	1	141 491	4	57 129	1	76 962	2	31 329	2	25 691	2	35 458	1	66 023
total	66	20	740 730	19	1 098 693	15	314 050	17	264 013	15	156 011	15	192 692	16	234 562

Table 2: Compared performances of the same GLUCOSE and PO-GLUCOSE variations on the same series of benchmarks. For each series, besides the number of timeouts (*#to*), the total number of clause checks performed (*checks*, given in millions) is listed. When several solvers timed out on the same instance, they were considered as having all needed the same amount of clause checks (the smallest amount amongst timed out solvers).

4.1 Analyzing efficiency of PO-CDCL

In this subsection, we will consider the default version of PO-GLUCOSE as described in [11] with a choice of the assertion level similar to its definition in a total order CDCL: amongst all candidate assertion levels, the most recently created one is picked. This default version is named *PO* in all tables and figures of this paper. Results of GLUCOSE with and without phase saving are labelled as *TO-phase* and *TO* respectively, *TO-phase* being the default GLUCOSE setting.

As expected, when a conflict occurs during a CDCL solving, there is in practice often a non-negligible quantity of instantiations between the assertion level and the conflict level. Therefore, on our formal verification instances, PO-GLUCOSE locally saves on average 15% of instantiations that would be deleted by a regular CDCL algorithm (they are located in decision levels instantiated after the assertion level but not depending on it). If we consider an entire solving trace, it however deletes on average approximately the same number of instantiations per conflict than the original GLUCOSE, as shown in Table 3. The efficiency of PO-GLUCOSE is thus not obtained by accumulating instantiations faster than with a total order; saved instantiations are likely to be deleted later. However, we will show that although instantiations are only saved temporarily, they can have a significant impact on the overall search.

Tables 1 and 2 show respectively the total time and clause checks needed to solve each benchmark family with this chronological heuristic, compared with performances of the two total order variants. Both versions of GLUCOSE have very contrasted results: the default version with phase saving clearly outperforms the version without phase saving on both satisfiable series, but conversely the version without phase saving performs better on 3 of the 4 unsatisfiable families.

When comparing solving time for each series separately, the performance of PO-GLUCOSE is generally close to the best performing GLUCOSE version and significantly better than the other (except on pipe_ooo_unsat_1.0, where it requires a little more time than the slowest GLUCOSE variant). It also never causes more than one additional timeout than the best performing GLUCOSE version. Thanks to this more balanced behaviour, it significantly outperforms both GLUCOSE with and without phase saving when considering the total solving time on all benchmarks, and manages to solve 4 to 5 more instances in the given time limit.

The cactus plots of Fig. 1 give a better view of the performances on individual instances. Top figures show how many satisfiable and unsatisfiable instances respectively can be solved within a given time limit. The top left figure indicates that PO-GLUCOSE manages to solve many instances very quickly (13 out of 20 are solved in less than 30 seconds each). When the time limit increases, it is eventually beaten by the default setup of GLUCOSE which is able to solve the 3 most difficult instances in a little less than 30 minutes while PO-GLUCOSE needs more time and fails to solve one of them within one hour. It however easily outperforms GLUCOSE without phase saving.

On unsatisfiable instances (top right figure), PO-Glucose considerably outperforms the default version of GLUCOSE with phase saving enabled, no matter what time limit is considered. Within one hour, it solves 5 more instances than default GLUCOSE. GLUCOSE without phase saving is however more efficient and slightly outperforms PO-

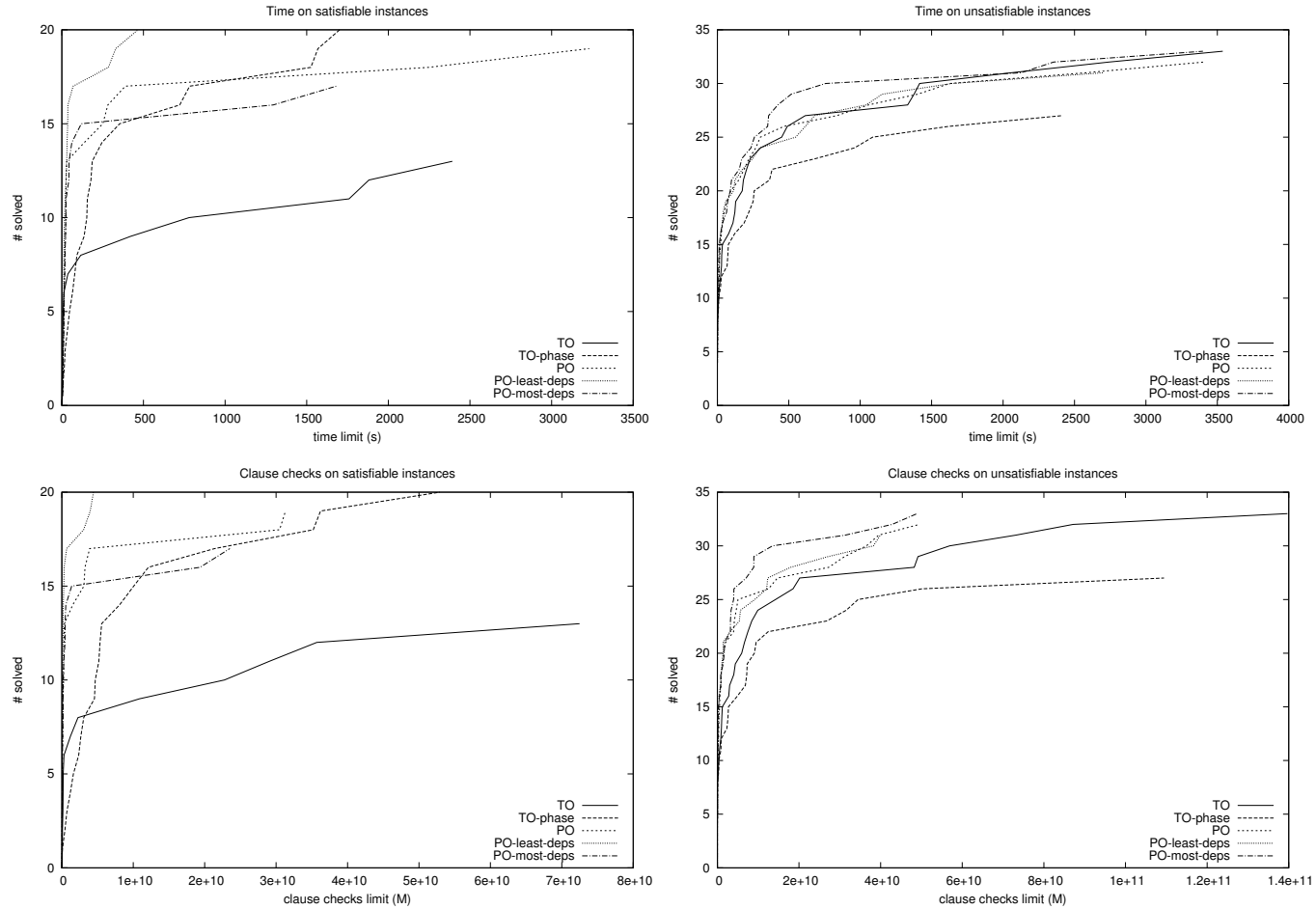


Fig. 1: Four cactus plots comparing the performances of two variants of GLUCOSE, without (*TO*) and with *(TO-phase*) phase saving, with the default PO-CDCL (*PO*) and PO-CDCL with two dependency-counting heuristics (*PO-least-deps, PO-most-deps*). Left plots compare the algorithms on the 20 satisfiable instances, right plots on the 46 unsatisfiable instances. x-axis measures the solving time on top plots and the clause checks performed on bottom plots.

GLUCOSE on high time limits, although the latter manages to solve more instances in 500 seconds or less.

Since about 40% of the solving time is an overhead due to handling level dependencies, the performance of PO-GLUCOSE is even better when the number of clause checks is considered. Bottom plots of Fig. 1 indicate that PO-GLUCOSE significantly outperforms both versions of GLUCOSE on most satisfiable and unsatisfiable instances. This observation is confirmed by Table 2, which shows that PO-GLUCOSE often requires dramatically less clause checks to solve the same amount of instances than GLUCOSE. Overall, PO-GLUCOSE solves more instances than both GLUCOSE implementations with twice to thrice less clause checks.

This efficiency can be explained by the effect of saved instantiations on the search. Table 4 shows the average amount of clause checks necessary to reach a conflict for various GLUCOSE and PO-GLUCOSE versions. This quantity is almost always dramatically lowered by PO-GLUCOSE, no matter what assertion heuristic is used. The instantiations saved by partial order backtracks, even if they are eventually deleted, seem to be often relevant and help reaching conflicts much faster. As each conflict prunes a part of the search space, partial order thus apparently dramatically improves this pruning, which obviously should help in proving unsatisfiability faster, but in also guiding the search in satisfiable instances towards branches of the search space containing models.

4.2 Assertion level heuristics

The default chronological assertion level choice used in the previous subsection was designed to remain as close as possible to the original CDCL algorithm and evaluate the efficiency gain that can be obtained solely by removing less instantiations during backtracks, without further modifying the search. However, we will show that this choice can significantly modify the way the search space is explored, and that particular heuristics can be used to further improve performances of PO-GLUCOSE.

Tests with the chronological assertion level choice showed that in 31% of the conflicts, there are several candidate assertion levels, and when it happens there are on average about 10 distinct candidate levels. The strategy used to choose the assertion level thus can potentially have a significant impact on the entire search. Since the primary goal of PO-CDCL is to save instantiations during backtracks, a straightforward local heuristic (named *PO-less-undos* in tables) consists in picking the candidate assertion level that will undo the least instantiations, i.e. that minimizes the quantity of variables located in decision levels depending on the candidate assertion level. However, according to Table 3, this strategy almost doesn't modify the average number of undos per conflict. Consequently, performances obtained with this heuristic are relatively close to results of the default chronological heuristic, as shown in Tables 1 and 2. This seems to indicate that the chronological heuristic already often picks assertion levels causing few uninstantiations.

Surprisingly, the opposite heuristic of picking the assertion level that will cause the most deletions (*PO-most-undos*) is much more interesting. Its performances on unsatisfiable instances are very close to performances of the chronological heuristic. However, as shown in Tables 1 and 2, it dramatically reduces the time and clause checks needed to solve satisfiable instances. pipe_sat_1.0 is solved about 3 times faster and with 7

family	#inst	TO	TO-phase	PO	PO-least-undos	PO-most-undos	PO-least-deps	PO-most-deps
pipe_sat_1.0	10	1 226	2 053	1 751	1 698	3 680	4 602	1 972
pipe_sat_1.1	10	1 099	1 726	1 957	1 599	3 834	4 983	1 691
pipe_unsat_1.0	13	885	968	1 050	1 046	1 241	1 244	970
pipe_unsat_1.1	14	1 124	1 054	1 096	1 066	1 284	1 316	974
pipe_ooo_unsat_1.0	9	648	679	660	648	685	693	624
pipe_ooo_unsat_1.1	10	784	610	749	718	781	755	741
average	11	972	1 172	1 204	1 129	1 867	2 185	1 151

Table 3: Comparison of the average number of instantiations undone at each backtrack by various solvers on some benchmark series. Solvers and benchmarks tested are the same as in Table 1.

series	#inst	TO	TO-phase	PO	PO-least-undos	PO-most-undos	PO-least-deps	PO-most-deps
pipe_sat_1.0	10	5 164	276	18	23	13	9	37
pipe_sat_1.1	10	2 999	16	11	17	6	10	25
pipe_unsat_1.0	13	1 214	1 499	21	28	18	10	39
pipe_unsat_1.1	14	203	1 271	19	21	13	9	19
pipe_ooo_unsat_1.0	9	99	13	7	8	5	4	7
pipe_ooo_unsat_1.1	10	25	1 283	9	9	6	6	8
average	11	1 546	805	14	19	11	8	23

Table 4: Comparison of the average number of clause checks (in millions) needed to reach a conflict by various solvers on some benchmark series. Solvers and benchmarks tested are the same as in Table 1. Note that the correlation between solving performances and the number of clause checks per conflict can be confirmed by comparing both total order versions *TO* and *TO-phase*: the best performing version on a benchmark series is always the one with the least checks per conflict.

times less clause checks than the best performing GLUCOSE version. pipe_sat_1.1 is solved more than 6 times faster and with 28 times less clause checks.

On these satisfiable series, as indicated by Table 3, the *most undos* heuristic deletes about twice more instantiations than default PO-GLUCOSE and both total order GLUCOSE implementations. The performance of this heuristic is likely due to this large amount of deletions, coupled to the frequent conflicts caused by partial order CDCL. Our intuition was that keeping as many instantiations as possible would help building a model of the instance faster, but apparently undoing as many instantiations as possible is more useful. It indeed certainly allows to skip unsatisfiable parts of the search space more quickly and to explore more various parts of this space.

Heuristics based on counting instantiations to be undone during the conflict have the drawback to be highly local, and consequently they generally don't reach their goal globally. Indeed, the choice of the assertion level doesn't only affect the current backtrack: dependencies are added between this level and all other levels involved in the conflict. These additional dependencies increase the likelihood for the chosen assertion level to be deleted in future conflicts. If the conflict clause involves n decision levels (not including the conflict level), the assertion level will have to depend on all other $n - 1$ levels, but some of these dependencies may already exist. Intuitively, picking the candidate assertion level which will entail the least new dependencies should tend to globally lower the average quantity of instantiations undone during a conflict. Conversely, we expect the opposite heuristic to delete more instantiations per conflict.

For some unexplained reason, it is exactly the opposite that happens. The *least dependencies* strategy causes even more uninstantiations than the *most undos* heuristic, causing a slight increase of solving time on unsatisfiable instances, but a further improvement of performances on satisfiable instances. Figure 1 shows that with this heuristic 17 of the 20 satisfiable instances are solved within 70 seconds, the 3 remaining instances being solved in less than 500 seconds each. In contrast, 11 instances require more than 100 seconds and 3 more than 1 500 seconds with the best performing GLUCOSE version.

On the other hand, the *most dependencies* heuristic performs poorly on satisfiable instances but very well on unsatisfiable instances. Figure 1 shows that it is by far the best tested solver in terms of checked clauses and that it even steadily outperforms the best GLUCOSE version on all time limits.

This performance is explained by a sensible decrease of the average number of undone instantiations per conflict compared to other PO-Glucose implementations, as shown in Table 3. In the case of unsatisfiable instances, undoing less instantiations seems to help focussing the search on the currently active part of the search space. Favorizing successive conflicts in related parts of the search space results in a more efficient pruning and ultimately requires less conflicts to prove unsatisfiability: regular GLUCOSE with and without phase saving need on average about 7 and 4,6 millions of conflicts respectively for solving unsatisfiable benchmarks. This number drops to between 2 and 2,7 millions of conflicts for previous PO-GLUCOSE variants, and down to 1,75 million with the *most dependencies* heuristic.

These dependencies-oriented heuristics and their contrasted efficiency suggest that on SAT problems with low decision level interdependencies, satisfiability solving can

be significantly improved by using totally different strategies depending on the actual satisfiability of the instance: if a model exists, it can be found easier if backtracks undo many instantiations, which helps exploring the search space more dynamically. In the unsatisfiable case, backtracks should at the opposite undo less instantiations to help focus the search on the currently active search space and prove unsatisfiability with less conflicts. Moreover, both types of strategies can be carried out by an appropriate choice of assertion levels in a partial order CDCL search.

Satisfiability of instances with sparse level dependencies can thus be very efficiently checked with PO-CDCL if the answer is known or speculated prior to solving. We think it should be possible to design a more balanced intermediate strategy that would perform significantly better than total order CDCL regardless of the instance satisfiability.

5 Conclusion

In this paper, we further analyzed the partial order CDCL algorithm and its behaviour on instances with sparse dependencies between decision levels. We showed that the instantiations saved by the less destructive backtrack of PO-CDCL often allow to discover conflicts dramatically faster, which helps to prune the search space more efficiently. This behaviour explains the good solving performances observed on tested instances. Moreover, we noticed the significant impact of the assertion level choice on the search and designed several heuristics for this choice. According to our observations, opposite strategies are relevant depending on whether the solved instance is or isn't satisfiable. A satisfying model of the problem can be found faster if the backtrack generally undoes large parts of the assignment, allowing quicker moves in the search space. Conversely, undoing a smaller average quantity of instantiations helps the solver to focus on the currently active part of the search space and leads faster to a proof of unsatisfiability. Finally, we showed that trying to locally control the amount of instantiations undone by each individual backtrack is not the most efficient method; heuristics that choose the assertion level according to the amount of level dependencies it introduces have a stronger influence on the average quantity of assignment deletions.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI 2009. pp. 399–404
2. Bhalla, A., Lynce, I., de Sousa, J.T., Marques-Silva, J.: Heuristic-based backtracking relaxation for propositional satisfiability. Journal of Automated Reasoning 35(1–3), 3–24 (2005)
3. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Communications of the ACM 5(7), 394–397 (1962)
4. Durairaj, V., Kalla, P.: Exploiting hypergraph partitioning for efficient boolean satisfiability. In: HLDVT 2004. pp. 141–146
5. Ginsberg, M.L.: Dynamic backtracking. Journal of Artificial Intelligence Research 1, 25–46 (1993)
6. Huang, J., Darwiche, A.: A structure-based variable ordering heuristic for SAT. In: IJCAI-03. pp. 1167–1172

7. Jégou, P., Terrioux, C.: Hybrid backtracking bounded by tree-decomposition of constraint networks. Artificial Intelligence 146(1), 43–75 (2003)
8. Li, W., van Beek, P.: Guiding real-world SAT solving with dynamic hypergraph separator decomposition. In: ICTAI 2004. pp. 542–548
9. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
10. McAllester, D.A.: Partial order backtracking. Research note, MIT (1993)
11. Monnet, A., Villemaire, R.: CDCL with less destructive backtracking through partial ordering. In: PAAR 2012. pp. 124–138
12. Monnet, A., Villemaire, R.: Scalable formula decomposition for propositional satisfiability. In: C^3S^2E 2010. pp. 43–52
13. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC 2001. pp. 530–535
14. Nadel, A., Ryvchin, V.: Assignment stack shrinking. In: SAT 2010. pp. 375–381
15. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: SAT 2007. pp. 294–299
16. Robertson, N., Seymour, P.D.: Graph minors. II. Algorithmic aspects of tree-width. Journal of Algorithms 7(3), 309–322 (1986)
17. Velev, M.N., Bryant, R.E.: Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. Journal of Symbolic Computation 35(2), 73–106 (2003)

Efficient Support for Mode-Directed Tabling in the YapTab Tabling System

João Santos and Ricardo Rocha

CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{jsantos,ricroc}@dcc.fc.up.pt

Abstract. Mode-directed tabling is an extension to the tabling technique that supports the definition of mode operators for specifying how answers are inserted into the table space. In this paper, we focus our discussion on the efficient support for mode directed-tabling in the YapTab tabling system. We discuss 7 different mode operators and explain how we have extended and optimized YapTab's table space organization to support them. Initial experimental results show that our implementation compares favorably with the B-Prolog and XSB state-of-the-art Prolog tabling systems.

1 Introduction

Tabling [1] is a recognized and powerful implementation technique that solves some limitations of Prolog's operational semantics in dealing with recursion and redundant sub-computations. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property*. Tabling consists of saving and reusing the results of sub-computations during the execution of a program and, for that, the calls and the answers to tabled subgoals are stored in a proper data structure called the *table space*. In a traditional tabling system, all the arguments of a tabled subgoal call are considered when storing answers into the table space. When a new answer is not a variant[1] of any answer that is already in the table space, then it is always considered for insertion. Therefore, traditional tabling systems are very good for problems that require storing all answers. *Mode-directed tabling* [2] is an extension to the tabling technique that supports the definition of selective criteria for specifying how answers are inserted into the table space. The idea of mode-directed tabling is to use *mode operators* to define what arguments should be used in variant checking in order to select what answers should be tabled.

In a traditional tabling system, to evaluate a predicate p/n using tabling, we just need to declare it as '$table\ p/n$'. With mode-directed tabling, tabled predicates are declared using statements of the form '$table\ p(m_1, ..., m_n)$', where the m_i's are mode operators for the arguments. Implementations of mode-directed tabling are already available in systems like ALS-Prolog [2] and B-Prolog [3], and a restricted form of mode-directed tabling can be also recreated in XSB Prolog by using *answer subsumption* [4].

[1] Two (answer or subgoal) terms are considered to be variant if they are the same up to variable renaming.

In this paper, we focus our discussion on the efficient implementation of mode directed-tabling in the YapTab tabling system [5], which uses *tries* [6] to implement the table space. Our implementation uses a more general approach to the declaration and use of mode operators and, currently, it supports 7 different modes: *index*, *first*, *last*, *min*, *max*, *sum* and *all*. To the best of our knowledge, no other tabling system supports all these modes and, in particular, the *sum* mode is not supported by any other system. Experimental results, using a set of benchmarks that take advantage of mode-directed tabling, show that our implementation compares favorably with the B-Prolog and XSB state-of-the-art Prolog tabling systems.

The remainder of the paper is organized as follows. First, we introduce some background concepts about tabling. Next, we describe the mode operators that we propose and we show some small examples of their use. Then, we introduce YapTab's table space organization and describe how we have extended it to efficiently support mode-directed tabling. At last, we present some experimental results and we end by outlining some conclusions.

2 Tabled Evaluation

In a traditional tabling system, programs are evaluated by storing answers for tabled subgoals in an appropriate data structure called the *table space*. Similar calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in the corresponding table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all similar calls.

Figure 1 illustrates the execution of a tabled program. The top left corner of the figure shows the program code and the top right corner shows the final state of the table space. The program defines a small directed graph, represented by two *edge/2* facts, with a relation of reachability, defined by a *path/2* tabled predicate. The bottom of the figure shows the evaluation sequence for the query goal *path(a,Z)*. Note that traditional Prolog would immediately enter an infinite loop because the first clause of *path/2* leads to a variant call to *path(a,Z)*.

First calls to tabled subgoals correspond to generator nodes (nodes depicted by white oval boxes) and, for first calls, a new entry, representing the subgoal, is added to the table space (step 0). Next, *path(a,Z)* is resolved against the first matching clause calling, in the continuation, *path(a,Y)* (step 1). Since *path(a,Y)* is a variant call to *path(a,Z)*, we do not evaluate the subgoal against the program clauses, instead we consume answers from the table space. Such nodes are called *consumer nodes* (nodes depicted by black oval boxes). However, at this point, the table does not have answers for this call, so the computation is suspended.

The only possible move after suspending is to backtrack and try the second matching clause for *path(a,Z)* (step 2). This originates the answer $\{Z=b\}$, which is then stored in the table space (step 3). At this point, the computation at node 1 can be resumed with the newly found answer (step 4), giving rise to one more answer, $\{Z=a\}$ (step 5). This second answer is then also inserted in the table space and propagated to the consumer node (step 6), which originates the answer $\{Z=b\}$ (step 7). This answer had already

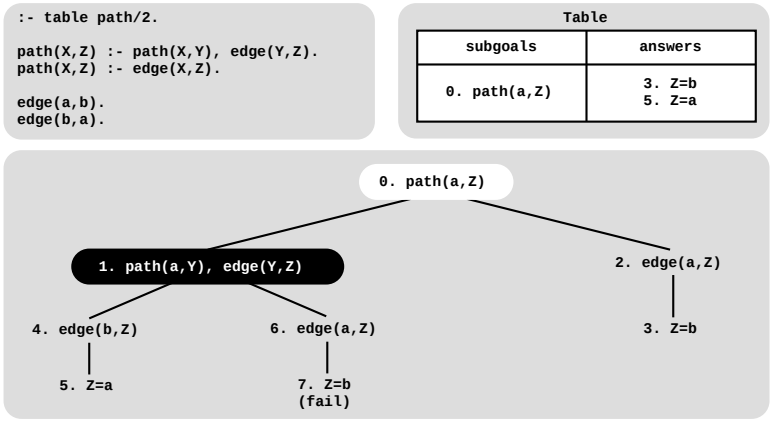


Fig. 1: An example of a tabled evaluation

been found at step 3. Tabling does not store duplicate answers in the table space and, instead, repeated answers *fail*. This is how tabling avoids unnecessary computations, and even looping in some cases. A new answer is inserted in table space only if it is not a variant of any answer that is already there. Since there are no more answers to consume nor more clauses left to try, the evaluation ends and the table entry for *path(a,Z)* can be marked as *completed*.

3 Mode-Directed Tabling

With mode-directed tabling, tabled predicates are declared using statements of the form '$table\ p(m_1, ..., m_n)$', where the m_i's are *mode operators* for the arguments. We have defined 7 different mode operators: *index*, *first*, *last*, *min*, *max*, *sum* and *all*. Arguments with modes *first*, *last*, *min*, *max*, *sum* or *all* are assumed to be output arguments and only *index* arguments are considered for variant checking. After an answer be generated, the system tables the answer only if it is *preferable*, accordingly to the meaning of the output arguments, than some existing variant answer. Next, we describe in more detail how these modes work and we show some examples of their use in the YapTab system.

3.1 Index/First/Last Mode Operators

Starting from the example in Fig. 1, consider now that we modify the program so that it also calculates the number of edges that are traversed in a path. Figure 2 illustrates the execution of this new program. As we can see, even with tabling, the program does not terminates. Such behavior occurs because there is a path with an infinite number of edges starting from *a*, thus not verifying the bounded term-size property necessary to ensure termination. In particular, the answers found at steps 3 and 7 and at steps 5 and 9 have the same answer for variable *Z* ({*Z*=*b*} and {*Z*=*a*}, respectively), but they are both inserted in the table space because they are not variants for variable *N*.

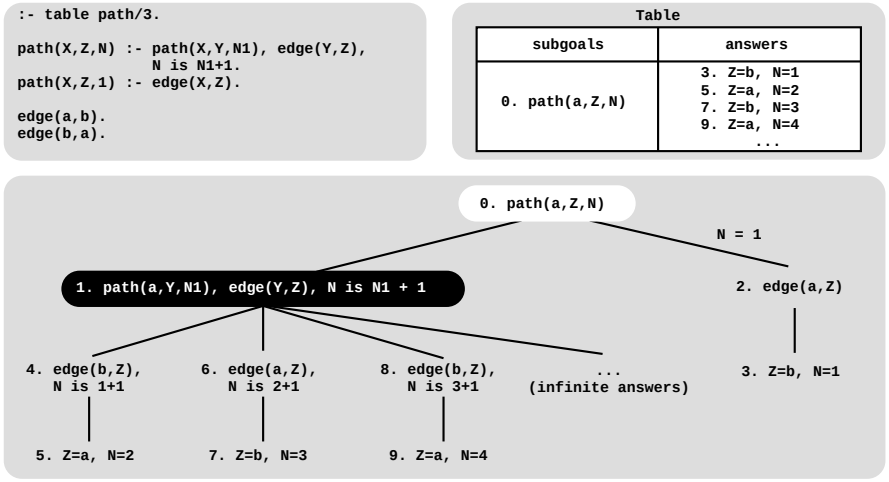


Fig. 2: A tabled evaluation with an infinite number of answers

Knowing that the problem with the program in Fig. 2 resides on the fact that the third argument generates an infinite number of answers, we can thus define the *path/3* predicate to have mode *path(index,index,first)*. The *index* mode means that only the given arguments must be considered for variant checking. The *first* mode means that only the first answer must be stored. By considering this declaration, the answer {*Z=b, N=3*} is no longer inserted in the table and execution fails. That happens because, with the *first* mode on the third argument, the answer {*Z=b, N=1*} found at step 3 is considered a variant of the answer {*Z=b, N=3*} found at step 7.

The *last* mode implements the opposite behavior of the *first* mode, i.e., it always stores the last answer being found and deletes the previous one, if any. The *last* mode has shown to be very useful for implementing problems involving Preferences [7] and Answer Subsumption [8].

3.2 Min/Max Mode Operators

The *min* and *max* modes allow to specify a selective criteria that stores, respectively, the minimal and maximal answers found for an argument. To better understand their behavior, Fig. 3 shows an example using the *min* mode. The program's goal is to compute the paths with the shortest distances. To do that, the *path/3* predicate is declared as *path(index,index,min)*, meaning that the third argument should store only the minimal answers for the first two arguments.

By observing the example in Fig. 3, we can see that the execution tree follows the normal evaluation of a tabled program and that the answers are stored as they are found. The most interesting part happens at step 8, where the answer {*Z=d, C=3*} is found. This answer is a variant of the answer {*Z=d, C=5*} found at step 6. In the previous example, with the *first* mode, the old answer would have been kept in the table. Here, as the new answer is minimal on the third argument, the old answer is replaced by the new answer.

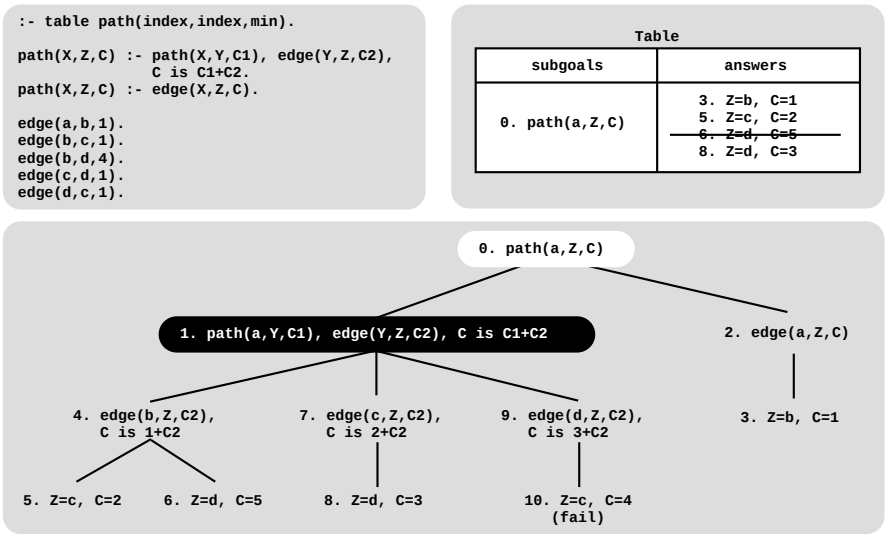


Fig. 3: Using the *min* mode to compute the paths with the shortest distances

The *max* mode works similarly, but stores the maximal answer instead. In any case, we must be careful when using these two modes as they may not ensure termination for programs without the bounded term-size property. For instance, this would be the case if, in the example of Fig.3, we used the *max* mode instead of the *min* mode.

3.3 Sum/All Mode Operators

Two other modes that can be useful are the *sum* and the *all*. The *sum* mode allows to sum all the answers for a given argument and the *all* mode allows to store all the answers for a given argument. Consider, for example, the program in Fig. 4 where the *path/3* predicate is declared as *path(index,index,min,all)* meaning that, for each path, we want to store the shortest distance of the path (the third argument) and, at the same time, we want to store the number of edges traversed, for all paths with the same minimal distances (the fourth argument).

The execution tree for the program in Fig. 4 is similar to the previous ones. The most interesting part happens when the answer {*Z=b, C=2, N=2*} is found at step 8. This answer is a variant of the answer found at step 3 and although both have the same minimal value (*C=2*), the new answer is still inserted in the table space since the number of edges (fourth argument) is different.

Notice that when the *sum* or *all* modes are used in conjunction with another mode, like the *min* mode in the example, it is important to keep in mind that the aggregation of answers made for the *sum* or *all* argument depends on the corresponding answer for the *min* argument. Consider, for example, that in the previous example we had found one more answer {*Z=b, C=1, N=4*}. In this case, the new answer would be inserted and the answers {*Z=b, C=2, N=1*} and {*Z=b, C=2, N=2*} would be deleted because the new answer corresponds to a shorter distance, as defined by the value *C=1* in the *min* argument.

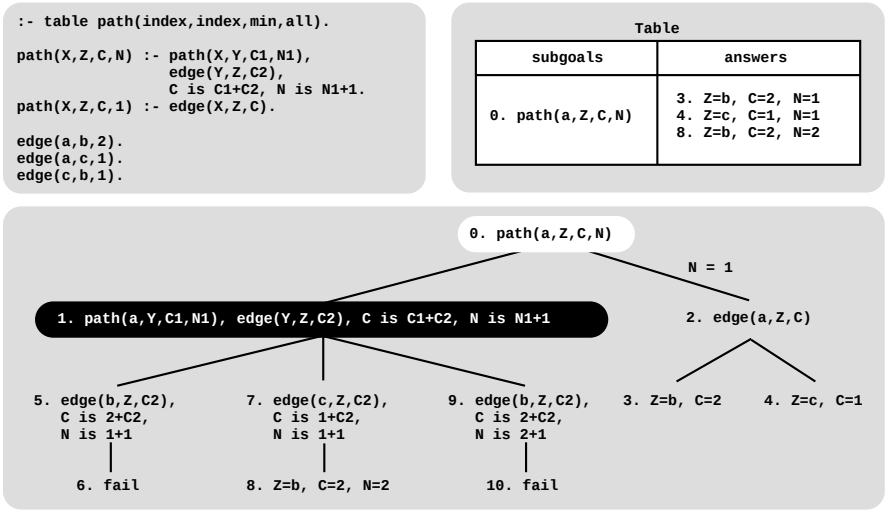


Fig. 4: Using the *all* mode to compute the paths with the shortest distances together with the number of edges traversed

3.4 Related Work

The ALS-Prolog [2] and B-Prolog [3] systems also implement mode-directed tabling using a very similar syntax. However, some mode operators have different names in those systems. For example, the *index*, *first* and *all* modes are known as +, - and @, respectively. The *sum* mode is not supported by any other system and B-Prolog also does not implement the *last* and *all* modes. The + (*index*) mode in B-Prolog is assumed to be an input argument, which means that it can only be called with ground terms. On the other hand, B-Prolog includes an extra mode, named *nt*, to indicate that a given argument should not be tabled and, thus, not considered to be inserted in the table space. B-Prolog also extends the mode-directed tabling declaration to include a *cardinality limit* that allows to define the maximum number of answers to be stored in the table space [3].

Mode-directed tabling can also be recreated in the XSB Prolog system by using *answer subsumption* [4]. XSB Prolog has two answer subsumption mechanisms. One is called *partial order answer subsumption* and can be used to mimic, in terms of functionality, the *min* and *max* modes. Consider that we want to use it with the program in Fig. 3 that computes the paths with the shortest distances. Then, we should declare the *path/3* predicate as $path(\_,\_,po(</2))$ meaning that the third argument will be evaluated using partial order answer subsumption, where the predicate $</2$ implements the partial order relation. The other two arguments are considered to be index arguments.

The other XSB's mechanism, called *lattice answer subsumption*, is more powerful and can be used to mimic, in terms of functionality, the other modes. To use it with the same example, we only need to change the *path/3* declaration to $path(\_,\_,lattice(min/3))$. Note that the *min/3* predicate must have three arguments. This is necessary since, with this mechanism, we can generate a third answer starting

from the new answer and from the answer stored in the table. For example, for the shortest path problem, the predicate *min/3* could be something like:

$$min(Old, New, Res) :- Old < New \rightarrow Res = Old\ ;\ Res = New.$$

4 Implementation

In this subsection, we describe the changes made to YapTab in order to support mode-directed tabling. We start by briefly presenting some background concepts about the table space organization in YapTab and then we discuss in more detail how we have extended it to efficiently support mode-directed tabling.

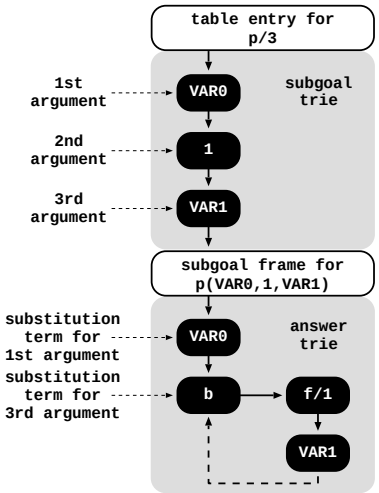
4.1 YapTab's Table Space Organization

Like we have seen, during the execution of a program, the table space may be accessed in a number of ways: (i) to find out if a subgoal is in the table and, if not, insert it; (ii) to verify whether a newly or preferable answer is already in the table and, if not, insert it; and (iii) to load answers from the tables.

With these requirements, a careful design of the table space is critical to achieve an efficient implementation. YapTab uses *tries* which is regarded as a very efficient way to implement the table space [6]. A trie is a tree structure where each different path through the *trie nodes* corresponds to a term described by the tokens labeling the traversed nodes. For example, the tokenized form of the term $path(X, 1, f(Y))$ is the sequence of 5 tokens $path/3$, VAR_0, 1, $f/1$ and VAR_1, where each variable is represented as a distinct VAR_i constant [9]. Two terms with common prefixes will branch off from each other at the first distinguishing token. Consider, for example, a second term $path(Z, 1, b)$, represented by the sequence of 4 tokens $path/3$, VAR_0, 1 and b. Since the main functor, token $path/3$, and the first two arguments, tokens VAR_0 and 1, are common to both terms, only one node will be required to fully represent this second term in the trie, thus allowing to save three nodes in this case.

YapTab's table design implements tables using two levels of tries. The first level, named *subgoal trie*, stores the tabled subgoal calls and the second level, named *answer trie*, stores the computed answers for a given call. More specifically, each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the predicate's subgoal trie. Each different subgoal call is then represented as a unique path in the subgoal trie, starting at the predicate's table entry and ending in a *subgoal frame* data structure, with the argument terms being stored within the path's nodes. The subgoal frame data structure acts as an entry point to the answer trie. Contrary to subgoal tries, answer trie paths hold just the substitution terms for the free variables that exist in the argument terms of the corresponding subgoal call [6].

An example for a tabled predicate $p/3$ is shown in Fig. 5. Initially, the table entry for $p/3$ points to an empty subgoal trie. Then, the subgoal $p(X, 1, Y)$ is called and three trie nodes are inserted to represent the arguments in the call:



one for variable X (VAR_0), a second for integer 1, and a last one for variable Y (VAR_1). Since the predicate's functor term is already represented by its table entry, we can avoid inserting an explicit node for $p/3$ in the subgoal trie. Then, the leaf node is set to point to a subgoal frame, from where the answers for the call will be stored. The example shows two answers for $p(X, 1, Y)$: {*X*=VAR_0, *Y*=*f*(VAR_1)} and {*X*=VAR_0, *Y*=*b*}. Since both answers have the same substitution term for argument X, they share the top node in the answer trie (VAR_0). For argument Y, each answer has a different substitution term and, thus, a different path is used to represent each.

When adding answers, the leaf nodes are chained in a linked list in insertion time order, so that the recovery may happen the same way. In Fig. 5, we can observe that the leaf node for the first answer (node VAR_1) points (dashed arrow) to the leaf node of the second answer (node b). To maintain this list, two fields in the subgoal frame data structure point, respectively, to the first and last answer of this list (for simplicity of illustration, these pointers are not shown in Fig. 5). When consuming answers, a consumer node only needs to keep a pointer to the leaf node of its last loaded answer, and consumes more answers just by following the chain. Answers are loaded by traversing the trie nodes bottom-up (again, for simplicity of illustration, such pointers are not shown in Fig. 5).

4.2 Mode-Directed Tabled Subgoal Calls

In YapTab, mode-directed tabled predicates are compiled by extending the table entry data structure to include a *mode array*, where the information about the modes is stored. In this mode array, the modes appear in the order in which the arguments are accessed, which can be different from their position in the original declaration. For example, *index* arguments must be considered first, irrespective of their position. Or, if using the *all* and *min* modes in a declaration, all *min* arguments must be considered before any *all* argument, since the *all* means that all answers must be stored, making meaningless the notion of being minimal in this case. As we will see in Section 4.3, changing the order is also strictly necessary to achieve an efficient implementation. In YapTab, the mode information is thus stored in the order mentioned below, together with the argument's position:

1. arguments with *index* mode;
2. arguments with *max* or *min* mode;
3. arguments with *all* mode;
4. argument (only one is allowed) with *sum* or *last* mode;
5. arguments with *first* mode.

Figure 6 shows an example for a *p(all,index,min)* mode-directed tabled predicate. The *index* mode is placed first in the mode array, then the *min* mode and last the *all* mode.

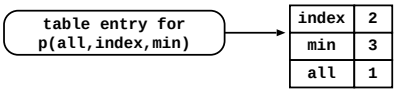


Fig. 6: Mode array

During tabled evaluation, new tabled subgoal calls are inserted in their own subgoal tries by following the order of the arguments in the call. With mode-directed tabling, we follow the order defined in the corresponding mode array. For example, consider again the mode-directed tabled predicate *p/3* as declared in Fig. 6 and the subgoal call *p(X,1,Y)*. Figure 7 shows the difference between the resulting subgoal tries with and without mode-directed tabling. The values in the mode array indicate that we should start by inserting first the second argument of the subgoal call (1), then the third argument (Y or VAR_0) and last the first argument (X or VAR_1).

The mode information is used when creating the subgoal frame associated with the subgoal call at hand. With mode-directed tabling, subgoal frames were extended to include a new array, named *substitution array*, where the mode information is stored, together with the number of free variables associated with each argument in the subgoal call. The argument's order is the same as in the mode array.

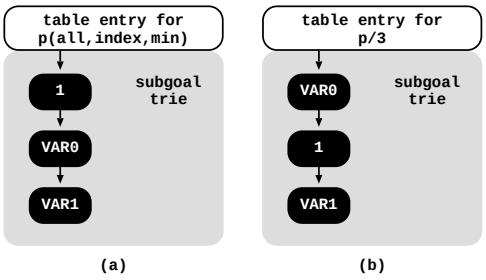


Fig. 7: Subgoal tries for *p(X,1,Y)* considering *p/3* declared (a) with and (b) without mode-directed tabling

Figure 8 shows the substitution array for the subgoal call *p(X,1,Y)*. The first position, corresponding to the argument with the constant 1, has no free variables and thus we store a 0 in the substitution array. The other two arguments are free variables and, thus, they have a 1 in the substitution array. It is possible to optimize the array by removing entries that have 0 variables and by joining contiguous entries with the same mode. As we will see next, the substitution array plays an important role in the process of inserting answers in the answer trie.

4.3 Mode-Directed Tabled Answers

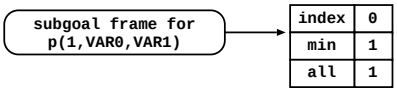


Fig. 8: Substitution array

Like in traditional tabling, tabled answers are only represented by the substitution terms for the free variables in the arguments of the corresponding subgoal call. However, for mode-directed tabling, when we are considering the substitution terms individually, it is important to know beforehand which mode applies to each, and for that, we use the information stored in the corresponding substitution array. Moreover, the substitutions *must be* con-

sidered in the same order that the variables they substitute have been inserted in the subgoal trie.

Consider again the substitution array for the subgoal call *p(X,1,Y)*. Now, if we find the answer *{X=f(a), Y=5}*, the first binding to be considered is *{Y=5}* with *min* mode and then *{X=f(a)}* with *all* mode. Since the answer trie is initially empty, both terms can be inserted as usual. Later, if another answer is found, for example, *{X=b, Y=3}*, we begin the insertion process by considering the binding *{Y=3}* with *min* mode. As there is already an answer in the table, we must compare both accordingly to the *min* mode. Since the new answer is preferable ($3 < 5$), the old answer must be *invalidated* and the new one inserted in the table. The invalidation process consists in: (a) deleting all intermediate nodes corresponding to the answers being invalidated; and (b) tagging the leaf nodes of such answers as *invalid nodes*. Invalid nodes are only deleted when the table is later completed or abolished. Figure 9 illustrates the aspect of the answer trie before and after the invalidation process.

Invalid nodes are opaque to subsequent subgoal calls, but can be still visible from the consumer calls already in evaluation. Hence, when invalidating a node, we may have consumers still pointing to it. By deleting leaf nodes, this would make consumers unable to follow the chain of answers. An alternative would be to traverse the stacks and update the consumers pointing to invalidated answers, but this could be a very costly operation.

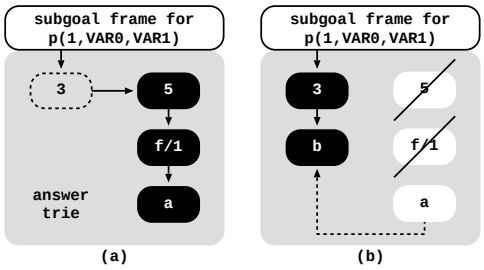


Fig. 9: Invalidating answers for *p(X,1,Y)* (a) before and (b) after the invalidation process

Notice also that the mode's order in the substitution array is crucial for the simplicity and efficiency of the invalidation process. When, at a given node N, we decide that an answer should be invalidated, the substitution array's order ensures that all nodes below node N (including N) are the ones we want to invalidate and that the upper nodes are the ones we want to keep. This might not be the case if we used the original order. For example, consider again the call *p(X,1,Y)* and the answers *{X=f(a), Y=5}* and *{X=b, Y=3}*. Figure 10 illustrates the invalidation process of these answers, if using the original declaration.

To detect that the second answer is preferable ($3 < 5$), we need to navigate in the trie until reaching the leaf node 5 for the first answer. Thus, the invalidation process may require deleting upper nodes (as the example in Fig. 10 shows) and/or traverse several paths to fully detect all preferable answers (this would be the case if we had two intermediate answers with the same minimal values, for instance *{X=f(a), Y=5}*

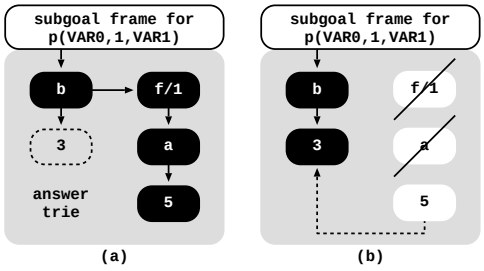


Fig. 10: Invalidating answers, without changing the insertion order, for *p(X,1,Y)* (a) before and (b) after the invalidation process

and *{X=h(c), Y=5}*), making therefore the invalidation process much more complex and costly.

4.4 Scheduling and Mode-Directed Tabling

In a tabled evaluation, there are several points where we may have to choose between continuing forward execution, backtracking, consuming answers, or completing subgoals. The decision on which operation to perform is determined by the scheduling strategy. The two most successful strategies are *batched scheduling* and *local scheduling* [10].

Batched scheduling evaluates programs in a depth-first manner as does the WAM. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues with forward execution. Only when all clauses have been resolved, the newly found answers will be forwarded to the consumers. Batched scheduling thus tries to delay the need to move around the search tree by batching the consumption of answers.

Local scheduling is an alternative scheduling strategy that tries to complete subgoals as soon as possible. The key idea is that whenever new answers are found, they are added to the table space, as usual, but execution fails. Local scheduling thus explores the whole search space for a tabled predicate before returning answers for forward execution.

To the best of our knowledge, YapTab is the only tabling system that supports the dynamic mixed-strategy evaluation of batched and local scheduling within the same evaluation [11]. This is very important, because for mode-directed tabled predicates, the ability of being able to use local evaluation can be crucial to correctly and/or efficiently support some modes.

This is the case for the *sum* mode. As it sums all the answers for a given argument, we might end with wrong results if we return partial results instead of aggregating them and only returning the aggregated result. Consider, for example, the two mode-directed tabled predicates $num\_links/2$ and $num\_nodes/1$ in Fig. 11 and the query goal $num\_nodes(N)$. If $num\_links/2$ is evaluated using local scheduling, we get the right result (*N=3*) but, with batched scheduling, we end with a wrong result (*N=6*). This occurs because, with batched evaluation, the $num\_links(\_,\_)$ call in the second clause of $num\_nodes/2$ succeeds 2 times for each $edge/2$ fact.

$$
\begin{array}{l}
:- table\ num\_links(index, sum).\\
num\_links(A, 0) :- edge(\_, A).\\
num\_links(A, 1) :- edge(A, \_).\\
\\
:- table\ num\_nodes(sum).\\
num\_nodes(0).\\
num\_nodes(1) :- num\_links(\_, \_).\\
\\
edge(a, b). \quad edge(a, c). \quad edge(b, c).
\end{array}
$$

Fig. 11: A cascade of two mode-directed tabled predicates using the *sum* mode

Batched evaluation can also yield useless computations for mode-directed tabled predicates (see Fig. 12). Consider, for example, a mode-directed tabled predicate $p/1$ declared as $p(max)$ and the query goal:

$$:- p(Max),\ do\_work(Max, Res).$$

With batched evaluation, the call to $do\_work(Max, Res)$ will be executed for each Max partial result computed by $p(Max)$, hence originating as many useless computations as the number of non-maximal results.

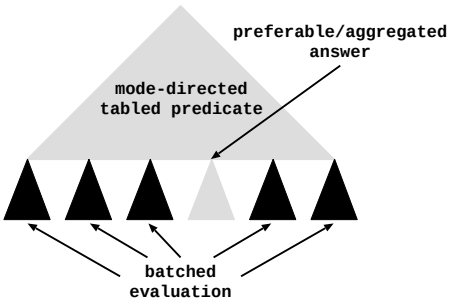


Fig. 12: Useless computations with batched evaluation

5 Experimental Results

In this section, we present some experimental results for a set of benchmarks that take advantage of mode-directed tabling. The environment for our experiment was a machine with a AMD FX(tm)-8150 8-core processor with 32 GBytes of main memory and running the Linux kernel 64 bits version 3.2.0. To put our results in perspective, we compare our implementation, on top of Yap Prolog (development version 6.3), with the B-Prolog (version 7.8 beta-6) and the XSB (version 3.3.6) systems, both using local scheduling. For XSB, we adapted the benchmarks to use lattice answer subsumption (as discussed in Section 3.4)[2]. For benchmarking, we used the following set of programs:

shortest(N) uses the *min* mode to determine all-pairs shortest paths in a graph representing the flight connections between the N busiest commercial airports in US[3].

shortest_first(N) uses the *first* mode to extend the all-pairs shortest paths program to also include the first justification for each shortest path.

shortest_all(N) uses the *all* mode to extend the all-pairs shortest paths program to also include all the justifications for each shortest path.

shortest_pref(N) uses the *last* mode to solve the all-pairs shortest paths program using Preferences [8].

knapsack(N) uses the *max* mode to determine the maximum number of items to include in a collection, from N weighted items, so that the total weight is equal to a given value.

lcs(N) uses the *max* mode to find the longest subsequence common to two different sequences of size N.

matrix(N) uses the *min* mode to implement the matrix chain multiplication problem that determines the most efficient way to multiply a sequence of N matrices.

[2] For programs using *min*/*max* modes, we also tried with partial order answer subsumption but, unexpectedly, we got worst results.

[3] `http://toreopsahl.com/datasets`

Table 1: Execution times, in milliseconds, for YapTab, B-Prolog and XSB and the respective overhead ratios when compared with YapTab's local evaluation

Programs	YapTab		B-Prolog	XSB
	Local	Batched		
shortest(300)	1,088	1,261 (1.16)	2.990 (2.37)	2,922 (2.69)
shortest(400)	1,544	1,785 (1.16)	4,216 (2.36)	4,321 (2.80)
shortest(500)	2,170	2,472 (1.14)	5,792 (2.34)	6,218 (2.87)
shortest_first(300)	1,394	2,641 (1.89)	3,225 (1.22)	5,013 (3.60)
shortest_first(400)	2,052	3,432 (1.67)	4,614 (1.34)	7,257 (3.54)
shortest_first(500)	2,866	4,228 (1.57)	7,400 (1.42)	10,328 (3.60)
shortest_all(300)	4,324	8,383 (1.94)	$n.a.$ (—)	61,803 (—)
shortest_all(400)	5,861	10,590 (1.81)	$n.a.$ (—)	122,985 (—)
shortest_all(500)	8,337	13,598 (1.63)	$n.a.$ (—)	239,451 (—)
shortest_pref(300)	2,882	4,241 (1.47)	$n.a.$ (—)	6,666 (2.31)
shortest_pref(400)	4,152	5,621 (1.35)	$n.a.$ (—)	9,932 (2.39)
shortest_pref(500)	5,773	7,473 (1.29)	$n.a.$ (—)	14,129 (2.45)
knapsack(1000)	1,013	998 (0.99)	837 (0.84)	2,684 (2.65)
knapsack(1500)	1,581	1,561 (0.99)	1,229 (0.79)	3,977 (2.52)
knapsack(2000)	2,037	2,040 (1.00)	1,582 (0.78)	5,473 (2.69)
lcs(1000)	1,196	1,416 (0.98)	2,900 (2.48)	3,060 (2.56)
lcs(1500)	2,768	3,560 (0.98)	5,784 (2.12)	7,128 (2.58)
lcs(2000)	4,864	6,053 (0.99)	10,116 (2.11)	13,338 (2.74)
matrix(100)	192	224 (1.17)	582 (2.60)	396 (2.06)
matrix(150)	925	1,076 (1.16)	2,549 (2.37)	1,610 (1.74)
matrix(200)	3,005	3,534 (1.18)	7,816 (2.21)	4,688 (1.56)
pagerank(1)	365	$n.a.$ (—)	$n.a.$ (—)	128,377 (—)
pagerank(16)	813	$n.a.$ (—)	$n.a.$ (—)	$> 10\ min$ (—)
pagerank(36)	1,260	$n.a.$ (—)	$n.a.$ (—)	$> 10\ min$ (—)
Average		(1.29)	(1.82)	(2.49)

pagerank(N) uses the *sum* mode to measure the rank values of web pages in a realistic dataset of web links called *search engines*[4], using N iterations.

Table 1 shows the execution times, in milliseconds, for running the benchmarks with YapTab, B-Prolog and XSB. In parentheses, it also shows the overhead ratios against YapTab with local evaluation. The execution times are the average of 3 runs. The entries marked with $n.a.$ correspond to programs using modes not available in B-Prolog. The ratios marked with (—) mean that we are *not considering* them in the average results (they correspond either to $n.a.$ entries or to execution times much higher than YapTab).

In general, the results show that, for all combinations of experiments and systems, there is no clear tendency showing that the overhead ratios increase or decrease as we increase the size of the corresponding set of programs.

Comparing the results for local and batched evaluation, they show that, on average, batched evaluation is around 29% worse than local evaluation. Batched evaluation gets

[4] `http://www.cs.toronto.edu/~tsap/experiments/download/download.html`

worse the more answers are inserted into the table space. This affects in particular the **shortest_first()**, **shortest_all()** and **shortest_pref()** set of programs, which confirms our discussion regarding the fact that batched evaluation is more suitable to useless computations.

Regarding the comparison with the other systems, the results obtained for YapTab clearly outperform those of B-Prolog and XSB. On average, B-Prolog and XSB are, respectively, around 1.82 and 2.49 times worse than YapTab using local evaluation.

Please note that for B-Prolog and XSB we do not include the performance of some programs into the average results. For B-Prolog, this is because these programs use the *all*, *last* and *sum* modes, which are not supported in B-Prolog. For XSB, the execution times for the **shortest_all()** and **pagerank()** are much higher than YapTab and including them would have distorted the comparison between the three systems. To the best of our knowledge, YapTab is thus the only system that supports the *all*, *last* and *sum* modes and handles them efficiently.

6 Conclusions

We discussed how we have extended and optimized YapTab's table space organization to provide engine support for mode-directed tabling. In particular, we presented how we deal with mode-directed tabled subgoal calls and answers and we discussed the role of scheduling in mode-directed tabled evaluations. Our implementation uses a more general approach to the declaration and use of mode operators and, currently, it supports 7 different modes. To the best of our knowledge, no other tabling system supports all these modes and, in particular, the *sum* mode is not supported by any other system. Experimental results on benchmarks that take advantage of mode-directed tabling, showed that our implementation clearly outperforms the B-Prolog and XSB state-of-the-art Prolog tabling systems. In particular, YapTab is the only system that efficiently handles programs that use the *all* mode. Further work will include extending our implementation to support multi-threaded mode-directed tabling.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects PEst (FCOMP-01-0124-FEDER-022701), HORUS (PTDC/EIA-EIA/100897/2008) and LEAP (PTDC/EIA-CCO /112158/2009). João Santos is funded by the FCT grant SFRH/BD/76307/2011.

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
2. Guo, H.F., Gupta, G.: Simplifying Dynamic Programming via Mode-directed Tabling. Software Practice and Experience **38**(1) (2008) 75–94

3. Zhou, N.F., Kameya, Y., Sato, T.: Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving. In: IEEE International Conference on Tools with Artificial Intelligence. Volume 2., IEEE Computer Society (2010) 213–218
4. Swift, T., Warren, D.S.: Tabling with Answer Subsumption: Implementation, Applications and Performance. In: European Conference on Logics in Artificial Intelligence. Number 6341 in LNAI, Springer-Verlag (2010) 300–312
5. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming **5**(1 & 2) (2005) 161–205
6. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54
7. Guo, H.F., Jayaraman, B., Gupta, G., Liu, M.: Optimization with Mode-Directed Preferences. In: 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM (2005) 242–251
8. Santos, J., Rocha, R.: Mode-Directed Tabling and Applications in the YapTab System. In: Symposium on Languages, Applications and Technologies. (2012) 25–40
9. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: International Joint Conference on Theory and Practice of Software Development. Number 668 in LNCS, Springer-Verlag (1993) 61–74
10. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1140 in LNCS, Springer-Verlag (1996) 243–258
11. Rocha, R., Silva, F., Santos Costa, V.: Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 250–264

Profiling Large Tabled Computations using Forest Logging

Terrance Swift

CENTRIA, Departamento de Informática, Faculdade de Ciencia e Tecnologia, Universidade Nova de Lisboa, Portugal **.

Abstract. Knowledge representation systems that are based on the well-founded semantics make use of HiLog, frame-based reasoning, defeasibility theories and other expressive features. These constructs can be compiled into Prologs that have good support for tabling, indexing and other extensions. However, the resources used for query evaluation by such systems can be unpredictable, due both to the power of the semantic features and to the declarative style typical of knowledge representation rules. In such a situation, users need to understand the overall structure of a computation and examine problematic portions of it. This problem, of *profiling* a computation, differs from debugging and justification which address why a given answer was or wasn't derived, and so profiling requires different techniques. In this paper we present a new technique called *forest logging* which has been used to profile large, heavily tabled computations. In forest logging, critical aspects of a tabled computation are logged; afterwards the log is loaded and analyzed. As implemented in XSB, forest logging slows down execution of practical programs by a small constant factor, and logs of tens or hundreds of millions of facts can be loaded and analyzed in minutes.

** Only abstract included in this version due to copyright restrictions.

On Comparing Alternative Splitting Strategies for Or-Parallel Prolog Execution on Multicores

Rui Vieira, Ricardo Rocha, and Fernando Silva

CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{revs,ricroc,fds}@dcc.fc.up.pt

Abstract. Many or-parallel Prolog models exploiting implicit parallelism have been proposed in the past. Arguably, one of the most successful models is *environment copying* for shared memory architectures. With the increasing availability and popularity of multicore architectures, it makes sense to recover the body of knowledge there is in this area and re-engineer prior computational models to evaluate their performance on newer architectures. In this work, we focus on the implementation of splitting strategies for or-parallel Prolog execution on multicores and, for that, we develop a framework, on top of the YapOr system, that integrates and supports five alternative splitting strategies. Our implementation shares the underlying execution environment and most of the data structures used to implement or-parallelism in YapOr. In particular, we took advantage of YapOr's infrastructure for incremental copying and scheduling support, which we used with minimal modifications. We thus argue that all these common support features allow us to make a first and fair comparison between these five alternative splitting strategies and, therefore, better understand their advantages and weaknesses.

1 Introduction

Detecting parallelism is far from a simple task, specially in the presence of irregular parallelism, but it is commonly left to programmers. Research effort has been made towards making specialized run-time systems more capable of transparently exploring available parallelism, thus freeing programmers from such cumbersome details. Prolog programs naturally exhibit *implicit parallelism* and are thus highly amenable for automatic exploitation.

One of the most noticeable sources of parallelism in Prolog programs is called *or-parallelism*. Or-parallelism arises from the simultaneous evaluation of a subgoal call against the clauses that match that call. When implementing or-parallelism, a main difficulty is how to efficiently represent the *multiple bindings* for the same variable produced by the parallel execution of alternative matching clauses. One of the most successful models is *environment copying* [1,2], that has been efficiently used in the implementation of or-parallel Prolog systems on shared memory architectures. Recent advances in computer architectures have made our personal computers parallel with multiple cores sharing the main memory. Multicores and clusters of multicores are now the norm and, although, many parallel Prolog systems have been developed in the past,

evaluating their performance or even the implementation of newer computational models specialized for the multicores is still open to further research.

Another major difficulty in the implementation of any parallel system is to design efficient *scheduling strategies* to assign computing tasks to workers waiting for work. A parallel Prolog system is no exception as the parallelism that Prolog programs exhibit is usually highly irregular. Achieving the necessary cooperation, synchronization and concurrent access to shared data structures among several workers during execution is a difficult task. For environment copying, scheduling strategies based on *bottommost dispatching of work* have proved to be more efficient than topmost strategies [3]. An important mechanism that suits bottommost strategies best is *incremental copying* [1], an optimized copy mechanism that avoids copying the whole stacks when sharing work. *Stack splitting* [4,5] is an extension to the environment copying model that provides a simple, clean and efficient method to accomplish work splitting among workers. It successfully splits the computation task of one worker in two complementary sets, and was thus first introduced aiming at distributed memory architectures [6,7].

In this work, we focus on the implementation of splitting strategies for or-parallel Prolog execution on multicore architectures and, for that, we present a framework, on top of the YapOr system [2], that integrates and supports five alternative splitting strategies. We used YapOr's original splitting strategy [2] and two splitting strategies from previous work [8], named *vertical* and *half splitting*, that split work based on choice points, together with the new implementation of two alternative stack splitting strategies, named *horizontal* [4] and *diagonal splitting* [7], in which the split is based on the unexplored alternative matching clauses. All implementations take full advantage of the state-of-the-art fast and optimized Yap Prolog engine [9] and share the underlying execution environment and most of the data structures used to implement or-parallelism in YapOr. In particular, we took advantage of YapOr's infrastructure for incremental copying and scheduling support, which we used with minimal modifications. We thus argue that all these common support features allow us to make a first and fair comparison between these five alternative splitting strategies and, therefore, better understand their advantages and weaknesses.

The remainder of the paper is organized as follows. First, we introduce some background about environment copying, stack splitting and YapOr's scheduler. Next, we describe the five alternative splitting strategies and discuss their major implementation issues in YapOr. We then present experimental results on a set of well-known benchmarks and advance some conclusions and further work.

2 Environment Copying

In the environment copying model, each worker keeps a separate copy of its own environment, thus enabling it to freely store assignments to shared variables without conflicts. Every time a worker shares work with another worker, all the execution stacks are copied to ensure that the requesting worker has the same environment state down to the search tree node[1] where the sharing occurs. To reduce the overhead of stack copying,

[1] At the engine level, a search tree node corresponds to a choice point in the stack.

an optimized copy mechanism called *incremental copy* [1] takes advantage of the fact that the requesting worker may already have traversed one part of the path being shared. Therefore, it does not need to copy the stacks referring to the whole path from root, but only the stacks starting from the youngest node common to both workers.

As a result of environment copying, each worker can proceed with the execution exactly as a sequential engine, with just minimal synchronization with other workers. Synchronization is mostly needed when updating scheduling information and when accessing shared nodes in order to ensure that unexplored alternatives are only exploited by one worker. Shared nodes are represented by *or-frames*, a data structure that workers must access, with mutual exclusion, to obtain the unexplored alternatives. All other data structures, such as the environment, the heap, and the trail do not require synchronization.

3 Stack Splitting

Stack splitting was first introduced to target distributed memory architectures, thus aiming to reduce the mutual exclusion requirements of environment copying when accessing shared nodes of the search tree. It accomplishes this by defining simple, clean and efficient work splitting strategies in which all available work is statically divided in two *complementary sets* between the sharing workers. In practice, stack splitting is a refined version of the environment copying model, in which the synchronization requirement was removed by the preemptive split of all unexplored alternatives at the moment of sharing. The splitting is such that both workers will proceed, each executing its branch of the computation, without any need for further synchronization when accessing shared nodes.

The original stack splitting proposal [4] introduces two strategies for dividing work: *vertical splitting*, in which the available choice points are alternately divided between the two sharing workers, and *horizontal splitting*, which alternately divides the unexplored alternatives in each available choice point. *Diagonal splitting* [7] is a more elaborated strategy that achieves a precise partitioning of the set of unexplored alternatives. It is a kind of mix between horizontal and vertical splitting, where the set of all unexplored alternatives in the available choice points is alternately divided between the two sharing workers. Another splitting strategy [10], which we named *half splitting*, splits the available choice points in two halves. Figure 1 illustrates the effect of these strategies in a work sharing operation between a busy worker P and an idle worker Q.

Figure 1(a) shows the initial configuration with the idle worker Q requesting work from a busy worker P with 7 unexplored alternatives in 4 choice points. Figure 1(b) shows the effect of vertical splitting, in which P keeps its current choice point and alternately divides with Q the remaining choice points up to the root choice point. Figure 1(c) illustrates the effect of half splitting, where the bottom half is for worker P and the half closest to the root is for worker Q. Figure 1(d) details the effect of horizontal splitting, in which the unexplored alternatives in each choice point are alternately split between both workers, with workers P and Q owning the first unexplored alternative in the even and odd choice points, respectively. Figure 1(e) describes the diagonal splitting strategy, where the unexplored alternatives in all choice points are alternately split

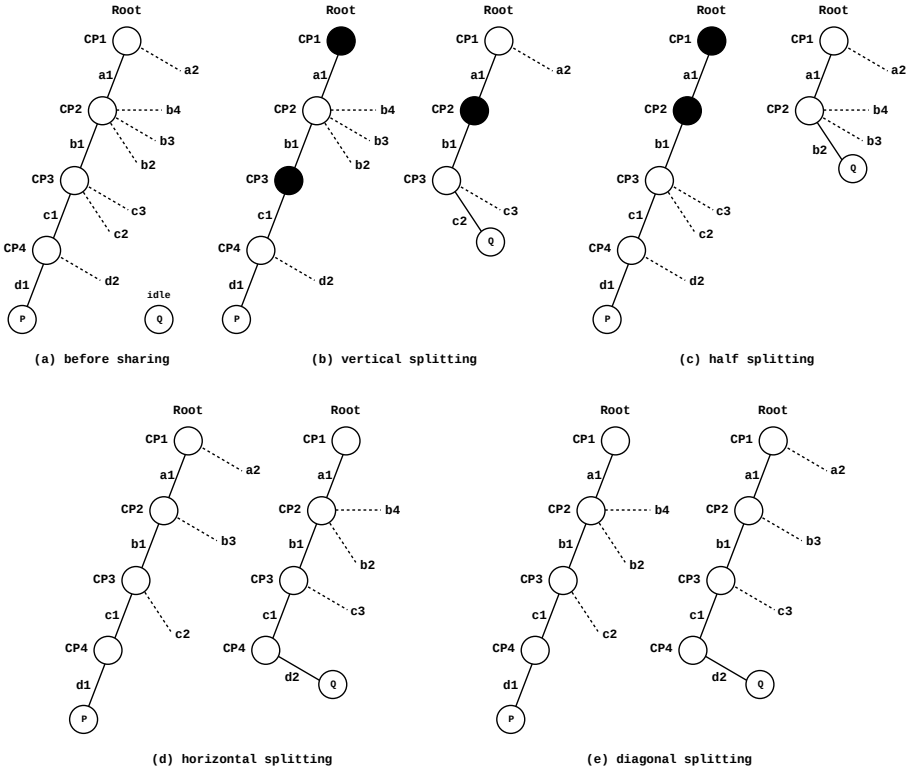


Fig. 1: Alternative stack splitting strategies

between both workers in such a way that, in the worst case, Q may stay with one more alternative than P. For all strategies, the corresponding execution stacks are first copied to Q, next both P and Q perform splitting, according to the splitting strategy at hand, and then P and Q are set to continue execution. As we will see, in some situations, there is no need for any copy at all, and a backtracking action is enough to place the requesting worker ready for execution.

4 YapOr's Scheduler and Original Splitting Strategy

We can divide the execution time of a worker in two modes: *scheduling mode* and *engine mode*. A worker enters in scheduling mode whenever it runs out of work and calls the scheduler to search for available work. As soon as it gets a new piece of work, it enters in engine mode and runs like a sequential engine.

4.1 Work Scheduling

In YapOr, when a worker runs out of work, first the scheduler tries to select a busy worker with excess of *work load* to share work. The work load is a measure of the

amount of unexplored alternatives in private nodes. There are two alternatives to search for busy workers in the search tree: search *below* or search *above* the current node where the idle worker is positioned. Idle workers always start to search below the current node, and only if they do not find any busy worker there, they search above. The main advantage of selecting a busy worker below instead of above is that the idle worker can request immediately the sharing operation, because its current node is already common to the busy worker, which avoids backtracking in the tree and undoing variable bindings.

When the scheduler does not find any busy worker with excess of work load, it tries to move the idle worker to a better position in the search tree. By default, the idle worker backtracks until it reaches a node where there is at least one busy worker below. Another option is to backtrack until reaching the node that contains all the busy workers below. The goal of these strategies is to distribute the idle workers in such a way that the probability of finding, as soon as possible, busy workers with excess of work below is substantially increased.

4.2 Work Sharing

Similarly to the Muse system[3], YapOr also follows a *bottommost work sharing strategy*. Whenever an idle worker *Q* makes a work request to a busy worker *P*, the work sharing operation is activated to *share all private nodes* of *P* with *Q*. *P* accepts the work request only if its work load is above a given *threshold value*. In YapOr, accomplishing this operation involves the following stages:

Sharing loop. This stage handles the sharing of *P*'s private nodes. For each private node, a new or-frame is allocated and the access to the unexplored alternatives, previously done through the `CP_alt` fields in the private choice points, is moved to the `OrFr_alt` fields in the new or-frames. All nodes have now a corresponding or-frame, which are sequentially chained through the fields `OrFr_next` and `OrFr_nearest_livenode`. The `OrFr_nearest_livenode` field is used to optimize the search for shared work. The membership field `OrFr_members`, which defines the set of workers that own or act upon a node, is also initialized to indicate that *P* and *Q* are sharing the corresponding choice points.

Membership update. Next, the old or-frames on *P*'s branch are updated to include the requesting worker *Q* in the membership field (frames starting from *P*'s current `top_or_frame` til *Q*'s `top_or_frame`). In order to delimit the shared region of the search tree, each worker maintains two important variables, named `top_cp` and `top_or_frame`, that point, respectively, to the youngest shared choice point and to the youngest or-frame[2].

Compute top or-frames. Finally, the new top or-frames in each worker are set, and since all shared work is available to both workers, both get the same `top_or_frame`. As we will see next, this is not the case for stack splitting, and

[2] Please note that the use of the naming *top* in these two variables can be confusing since, due to historical reasons, it refers to the top of the choice-point stack (where the root node is at the bottom) and not to the top of the search tree (where the root node is at the top). Despite this naming, our discussion keeps following a search tree approach with the root node always at the top.

the top_or_frame variable of *Q* is set accordingly to the splitting strategy being considered.

5 Supporting Alternative Splitting Strategies in YapOr

Extending YapOr to support different stack splitting strategies required some modifications to the way unexplored alternatives are accessed. In more detail:

- With stack splitting, each worker has its own work chaining sequence. Hence, the control and access to the unexplored alternatives returned to the CP_alt choice point fields and the OrFr_alt and OrFr_nearest_livenode or-frame fields were simply ignored.
- For the vertical and half splitting strategies, the OrFr_nearest_livenode field was recovered as a way to implement the chaining sequence of choice points. At work sharing, each worker adjusts its OrFr_nearest_livenode fields so that two separate chains are built corresponding to the intended split of the work.
- In order to reuse YapOr's infrastructure for incremental copying and scheduling support, the or-frames are still chained through the OrFr_next fields and still use the OrFr_member fields for work scheduling.

Next, we detail the implementation of the vertical, half, horizontal and diagonal splitting strategies as well as the incremental copy technique.

5.1 Vertical Splitting

The vertical splitting strategy follows a pre-determined work splitting scheme in which the chain of available choice points is alternately divided between the two sharing workers. At the implementation level, we use the OrFr_nearest_livenode field in order to generate two alternated chain sequences in the or-frames, and thus divide the available work in two independent execution paths. Workers can share the same or-frames but they have their own independent path without caring for the or-frames not assigned to them. Figure 2 presents the pseudo-code that implements the work sharing procedure for vertical splitting.

The work sharing procedure starts from *P*'s youngest choice point (register B) and traverses all *P*'s private choice points to create a corresponding or-frame by calling the alloc_or_frame() procedure. In Fig. 2, the current_fr, next_fr and nearest_fr variables represent, respectively, the or-frame allocated in the current step, the or-frame allocated in the previous step, which is used to link to the current or-frame by the OrFr_next field, and the or-frame allocated before the next_fr, which is used as a double spaced frame marker in order to initiate the OrFr_nearest_livenode fields. For the youngest choice point, the or-frame is initialized with just the owning worker *P* in the membership field. The other or-frames are initialized with both workers *P* and *Q*.

Next, follows the connection with the older and already stored or-frames. Here, consideration must be given to the condition of *P*'s current top_or_frame. If it is

```
next_fr = NULL
nearest_fr = NULL
current_cp = B  // B points to the youngest choice point
while (current_cp != top_cp)  // loop until the youngest shared choice point
  current_fr = alloc_or_frame(current_cp)
  add_member(P, OrFr_member(current_fr))
  if (next_fr)
    OrFr_next(next_fr) = current_fr
    add_member(Q, OrFr_member(current_fr))
  if (nearest_fr)
    OrFr_nearest_livenode(nearest_fr) = current_fr
  nearest_fr = next_fr
  next_fr = current_fr
  current_cp = CP_b(current_cp)  // next choice point on stack

// connecting with the older or-frames
if (next_fr)
  if (top_or_frame == root_frame)
    OrFr_nearest_livenode(next_fr) = DEAD_END
  else
    OrFr_nearest_livenode(next_fr) = top_or_frame
  OrFr_next(next_fr) = top_or_frame
if (nearest_fr)
  if (top_or_frame == root_frame)
    OrFr_nearest_livenode(nearest_fr) = DEAD_END
  else
    OrFr_nearest_livenode(nearest_fr) = top_or_frame

// continuing vertical splitting
if (next_fr = NULL)
  current_fr = top_or_frame
nearest_fr = OrFr_nearest_livenode(current_fr)
while (nearest_fr != DEAD_END)
  OrFr_nearest_livenode(current_fr) = OrFr_nearest_livenode(nearest_fr)
  current_fr = nearest_fr
  nearest_fr = OrFr_nearest_livenode(current_fr)
```

Fig. 2: Work sharing with vertical splitting

the root or-frame, the `OrFr_nearest_livenode` fields of the new or-frames are assigned to a `DEAD_END` value, which marks the ending point for unexplored work. Otherwise, they are assigned to *P*'s current `top_or_frame`.

Finally, we need to decide where to continue the vertical splitting algorithm for the older shared nodes. If no private work was shared, which means that we are only sharing work from the old shared nodes, the starting or-frame is *P*'s current `top_or_frame`. Otherwise, if some new or-frame was created, the starting or-frame is the last created frame in the sharing loop stage, which was connected to *P*'s current `top_or_frame` in the previous step. Either way, this serves the decision to elect the or-frame where the continuation of vertical splitting, guided through the `OrFr_nearest_livenode` field, should continue. The procedure then traverses the old shared frames until a `DEAD_END` is reached and, at each frame, lies a reconnection process of the `OrFr_nearest_livenode` field.

5.2 Half Splitting

The half splitting strategy partitions the chain of available choice points in two consecutive and almost equally sized parts, which are chained through the `OrFr_nearest_livenode` field of the corresponding or-frames. For that, the choice points are numbered sequentially and independently per worker to allow the calculation of the *relative depth* of the worker's assigned choice points. In order to support this numbering of nodes, a new *split counter* field, named `CP_sc`, was introduced in the choice point structure. Figure 3 presents the pseudo-code that implements work sharing with horizontal splitting.

```
// updating the split counter
current_cp = B  // B points to the youngest choice point
split_number = CP_sc(current_cp) / 2
while (CP_sc(current_cp) != split_number + 1)
  CP_sc(current_cp) = CP_sc(current_cp) - split_number
  current_cp = CP_b(current_cp)  // next choice point on stack
CP_sc(current_cp) = 1  // middle choice point

// assign the remaining choice points to the requesting worker
middle_fr = CP_or_fr(current_cp)
if (middle_fr)
  OrFr_nearest_livenode(middle_fr) = DEAD_END
  current_fr = top_or_frame  // top_or_frame points to the youngest or-frame
  while (current_fr != middle_fr)
    remove_member(Q, OrFr_member(current_fr))
    current_fr = OrFr_next(current_fr)
else
  // sharing loop stage
```

Fig. 3: Work sharing with half splitting

The work sharing procedure starts from *P*'s youngest choice point and updates the split counter on half of the choice points, in decreasing order, until reaching the *middle choice point* in *P*'s initial partition, which gets a split counter value of 1. These are the half choice points that, after sharing, will be still owned by *P*. The other half will be assigned to the requesting worker *Q*.

After updating the split counter, we can distinguish two different situations. The first situation occurs when there are more old shared choice points than private in *P*'s branch, in which case the middle choice point is already assigned with an or-frame. Thus, there is no need for the sharing loop stage, the middle frame is assigned to a `DEAD_END`, to mark the end of *P*'s newly assigned work, and the requesting worker *Q* is excluded from all or-frames from the top frame til the middle frame. The second situation occurs when the middle choice point is private, in which case the remaining choice points are updated to belong to *Q*, which includes allocating and initializing the corresponding or-frames.

5.3 Horizontal Splitting

In the horizontal splitting strategy, the unexplored alternatives are alternately divided in each choice point. For that, the choice points include an extra field, named `CP_offset`, that marks the offset of the next unexplored alternative belonging to the choice point. When allocating a private choice point, `CP_offset` is initialized with a value of 1, meaning that the next alternative to be taken has a displacement of 1 in the list of unexplored alternatives. This is the usual and expected behavior for private choice points.

When sharing work, we follow YapOr's default splitting strategy where a new or-frame is allocated for each private choice point in *P* and then all or-frames are updated to include the requesting worker *Q* in the membership field. Next, to implement the splitting process, we double the value of the `CP_offset` field in each shared choice point, meaning that the next alternative to be taken in the choice point is displaced two positions relatively to the previous value. Finally, we adjust the first alternative at each choice point for the workers *P* and *Q*. Recall from Fig. 1 that P must own the first unexplored alternative in the even choice points and Q the first unexplored alternative in the odd choice points. Figure 4 shows the pseudo-code for this procedure.

```
// the sharing worker P starts the adjustment
if (sharing worker) adjust = TRUE else adjust = FALSE
current_cp = top_cp
while(current_cp != root_cp)  // loop until the root choice point
  alt = CP_alt(current_cp)
  if (alt != NULL)
    offset = CP_offset(current_cp)
    CP_offset(current_cp) = offset * 2
    if (adjust)
      CP_alt(current_cp) = get_next_alternative(alt, offset)
  current_cp = CP_b(current_cp)  // next choice point on stack
  adjust = !adjust
```

Fig. 4: Work sharing with horizontal splitting

5.4 Diagonal Splitting

Diagonal splitting is an alternative strategy that implements a better overall distribution of unexplored alternatives between workers. Diagonal splitting is based on the alternated division of *all* alternatives, regardless of the choice points they belong to. This strategy also follows YapOr's default splitting strategy and uses the same offset multiplication approach as presented for horizontal splitting, but takes into account the number of unexplored alternatives in a choice point to decide how the partitioning will be done in the next choice point.

When a first choice point with an odd number of alternatives (say $2n + 1$) appears, the worker that must own the first alternative (say Q) is given $n + 1$ alternatives and the other (say P) is given n. The workers then alternate and, in the next choice point,

P starts the partitioning. When more choice points with an odd number of alternatives appear, the split process is repeated. At the end, Q and P may have the same number of unexplored alternatives or, in the worst case, Q may have one more alternative than P. The pseudo-code for this procedure is shown next in Fig. 5.

```
// the sharing worker P starts the adjustment
if (sharing worker) adjust = TRUE else adjust = FALSE
current_cp = top_cp
while(current_cp != root_cp)  // loop until the root choice point
  alt = CP_alt(current_cp)
  if (alt != NULL)
    offset = CP_offset(current_cp)
    CP_offset(current_cp) = offset * 2
    if (adjust)
      CP_alt(current_cp) = get_next_alternative(alt, offset)
    n_alts = number_of_unexplored_alternatives(alt) / offset
    if (n_alts mod 2 != 0)  // workers alternate
      adjust = !adjust
  current_cp = CP_b(current_cp)  // next choice point on stack
```

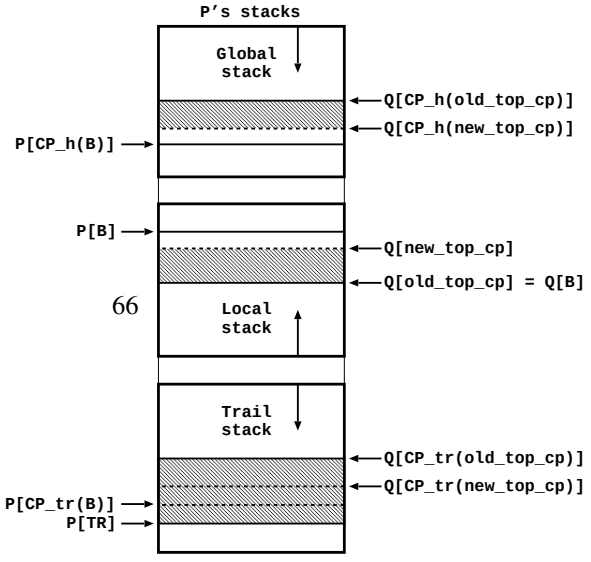
Fig. 5: Work sharing with diagonal splitting

5.5 Incremental Copy

In YapOr's original implementation, the incremental copy process copies everything in *P*'s stacks that is missing in *Q*. With stack splitting, it only copies the segments between *Q*'s `top_cp` before and after sharing for the global and local stacks. For the trail stack, the copy is the same since this is necessary to correctly implement the *installation phase* [2], where Q installs from P the bindings made to variables belonging to the common segments not copied from P.

Figure 6 illustrates the stack segments to be copied with incremental copy. For vertical splitting, if *P* has private work, Q's `new_top_cp` is assigned with the second choice point in *P*'s choice point set (`P[CP_b(B)]`). If there is no private work, the `new_top_cp` is assigned with the choice point corresponding to the or-frame pointed by `P[OrFr_nearest_livenode(CP_or_fr(old_top_cp))]`. For half splitting, the `new_top_cp` is always assigned with the choice point denoted by `P[CP_b(middle_cp)]`. For the horizontal and diagonal splitting, the assigning ranges are similar to YapOr's original implementation.

We next discuss the situations where Q's new `top_or_frame`, assigned during sharing, is older than Q's `top_or_frame` before sharing. In such case, Q does not copy any segment



from P and only needs to move up in the search tree in order to be consistent with the new assigned `top_or_frame`. In this movement, we may have to update the or-frames corresponding to the backtracked path by removing Q from the membership fields and by executing a *checking phase*. The checking phase is necessary to avoid incoherent values in the `CP_alt` fields in Q's choice points not copied from P. For half splitting, it also avoids incoherent values in the split counter fields for Q's choice points not copied from P. We can say that such incoherency can be caused by the independent work sharing operations with different workers that make the common (not copied) stack segments of P and Q, to be inconsistent in Q.

6 Experimental Results

In this section, we evaluate and compare the performance of the five splitting strategies on a set of well-known benchmarks. The environment for our experiments was a multi-core machine with 4 AMD Six-Core Opteron TM 8425 HE (2100 MHz) chips (24 cores in total) and 64 GB of DDR-2 667MHz RAM, running Linux (kernel 2.6.31.5-127 64 bits) with Yap Prolog 6.3.2. The machine was running in multi-user mode, but no other users were using it. For the benchmarks, we used the following set of programs:

cubes(N) a program that consists of stacking N colored cubes in a column in such a way that no color appears twice in the same column for each side.

ham(N) a program for finding all the Hamiltonian cycles in a graph with N nodes, with each node connected to 3 other nodes.

magic(N) a program to solve the Rubik's magic cube problem in N steps.

maze(N) a program that solves a maze problem in N steps by moving an empty square in a 4x4 grid.

nsort(N) a program for ordering a list of N elements using a naive algorithm and starting with the list inverted.

queens(N) a program to solve the N-queens problem that analyzes the board state at every step.

puzzle a program that solves a puzzle problem where the diagonals must add up to the same amount.

All benchmarks find all the solutions for the given problem by simulating an automatic failure whenever a new solution is found. Each benchmark was executed 10 consecutive times and the results are the average of those executions.

We start by measuring the cost of the parallel strategies over the sequential system. Table 1 presents the execution times, in seconds, for the set of benchmark programs, when using the sequential version of Yap and the respective ratios when using the several parallel models with one worker. In general, for all models, YapOr overheads result from handling the work load register and from operations that (i) verify whether the youngest node is shared or private, (ii) check for sharing requests, and (iii) check for backtracking messages due to cut operations.

Table 1: Execution times, in seconds, for Yap's sequential model and the respective overhead ratios for YapOr running 1 worker with YapOr's original splitting strategy (OS), vertical splitting (VS), half splitting (½S), horizontal splitting (HS) and diagonal splitting (DS).

Programs	**Yap**	**YapOr / Yap**				
		OS	**VS**	**½S**	**HS**	**DS**
cubes(7)	0.200	1.050	1.080	1.070	1.110	1.135
ham(26)	0.350	1.169	1.180	1.177	1.094	1.100
magic(6)	5.102	1.045	1.036	1.005	1.245	1.252
magic(7)	45.865	1.051	1.021	1.007	1.251	1.261
maze(10)	0.623	1.064	1.050	1.050	1.273	1.207
maze(12)	10.558	1.057	1.041	1.035	1.268	1.214
nsort(10)	2.775	1.124	1.155	1.096	1.074	1.072
nsort(12)	368.862	1.128	1.074	1.057	1.081	1.082
queens(11)	1.216	1.039	1.234	1.051	1.036	1.107
queens(13)	47.187	1.025	1.165	1.053	1.043	1.039
puzzle	0.153	1.157	1.235	1.144	1.176	1.157
Average		1.083	1.116	1.068	1.150	1.148

Results in Table 1 show that for these set of benchmarks, YapOr's overhead with each of the splitting strategies is small, between 6.8% and 15%. This is in-line with the overheads observed previously for YapOr and some of the splitting strategies [2,11,8].

Next, we assessed the performance of the or-parallel models, by running YapOr with a varying number of workers, up to 24, although for simplicity here we only show results for 16 and 24 workers. For fairness in the comparison of all strategies, we use the sequential execution times as the base execution times, instead of considering the base execution times with 1 worker for each strategy. In this way, the speedups do reflect real gains from sequential execution times. The results are shown in Tables 2 and 3 and the best speedup value among all strategies, which corresponds to the fastest execution times, for each benchmark, is marked with a gray background color.

From Table 2 we can observe the overall performance of all strategies without resorting to incremental copy optimization. The results show reasonably good speedups with exception for half splitting. With 24 workers, YapOr's original splitting shows the best performance, followed by vertical splitting and then horizontal and diagonal split-

Table 2: Speedups for YapOr running 16 and 24 workers with YapOr's original splitting strategy (OS), vertical splitting (VS), half splitting (½S), horizontal splitting (HS) and diagonal splitting (DS) without the incremental copy technique.

Programs	16 Workers					24 Workers				
	OS	**VS**	**½S**	**HS**	**DS**	**OS**	**VS**	**½S**	**HS**	**DS**
cubes(7)	6.45	4.65	0.61	5.26	5.12	6.66	3.92	0.46	4.76	4.54
ham(26)	6.14	4.86	2.34	4.11	5.14	6.36	4.79	2.07	3.97	5.14
magic(6)	14.33	14.25	8.35	11.67	11.70	20.40	19.77	7.76	16.51	16.35
magic(7)	14.97	15.51	12.18	12.29	12.31	22.24	22.96	16.17	18.39	18.43
maze(10)	9.58	10.74	4.82	7.78	7.98	11.32	11.98	4.20	9.16	8.41
maze(12)	14.44	15.06	11.55	12.50	12.56	21.03	21.81	14.89	17.80	17.68
nsort(10)	10.63	11.37	9.91	9.94	10.16	13.73	12.50	12.06	12.50	12.33
nsort(12)	14.37	14.71	14.72	14.43	14.52	21.16	21.47	21.62	20.93	20.78
queens(11)	12.66	7.84	1.68	11.05	11.15	16.21	8.94	1.60	13.07	12.93
queens(13)	15.66	14.05	4.10	15.08	15.16	22.14	20.54	4.12	22.20	22.42
puzzle	3.82	2.21	2.25	3.00	3.12	3.73	1.91	1.45	2.59	2.68
Average	11.19	10.48	6.59	9.74	9.90	15.00	13.69	7.85	12.90	12.88

ting with minimal differences. For some benchmarks, such as the **cubes** and **queens** benchmarks, half splitting does pretty badly.

Table 3 shows the overall performance for all strategies, but now using the incremental copying optimization. The performance for all strategies improve significantly for all benchmarks. Again, half splitting is the worst performing strategy, on average, it performs about 14% less than the best performing strategy with 24 workers. Another observation is that vertical, horizontal and diagonal splitting perform slightly close to the original YapOr. The best overall performance with 16 and 24 workers is achieved with vertical splitting.

Instead of using the sequential execution times as the base reference, if one uses the execution times with 1 worker for each strategy, then the average speedups with incremental copying and 24 workers for the original, vertical, horizontal and diagonal splitting were very close and above 20.

7 Conclusions and Further Work

We have presented the integration of five alternative splitting strategies on top of the YapOr system for or-parallel Prolog execution on multicores. Our implementation shares the underlying execution environment and most of the data structures used to implement or-parallelism in YapOr.

Experimental results, on a multicore machine with 24 cores, showed that clearly incremental copying optimization pays off in improving real performance in all strategies. The results for all strategies are reasonably good and the average speedups over

Table 3: Speedups for YapOr running 16 and 24 workers with YapOr's original splitting strategy (OS), vertical splitting (VS), half splitting (½S), horizontal splitting (HS) and diagonal splitting (DS) with the incremental copy technique.

Programs	16 Workers					24 Workers				
	OS	**VS**	**½S**	**HS**	**DS**	**OS**	**VS**	**½S**	**HS**	**DS**
cubes(7)	8.00	13.33	6.45	13.33	12.50	13.33	14.28	4.00	16.66	15.38
ham(26)	10.00	10.29	7.95	10.00	11.29	9.45	7.60	4.48	7.14	9.45
magic(6)	14.96	15.46	15.27	12.41	12.47	22.08	22.87	22.77	18.41	18.41
magic(7)	15.15	15.64	15.46	12.52	12.50	22.63	23.40	22.96	18.67	18.78
maze(10)	13.54	15.19	14.83	12.46	12.71	18.32	22.25	21.48	18.32	18.87
maze(12)	15.12	15.59	15.25	13.18	13.46	22.36	23.30	22.75	19.73	19.95
nsort(10)	14.15	14.60	14.60	14.15	14.08	20.25	20.70	21.34	19.96	20.40
nsort(12)	14.18	14.36	14.43	14.04	14.26	21.59	22.28	22.16	21.69	21.85
queens(11)	14.65	13.66	9.57	14.82	14.82	20.26	17.62	6.75	20.26	20.96
queens(13)	15.75	14.51	13.87	15.35	15.32	23.44	21.60	15.90	22.99	22.91
puzzle	9.00	10.20	11.76	11.76	11.76	9.56	10.20	15.30	10.92	12.75
Average	13.13	13.89	12.68	13.09	13.20	18.48	18.74	16.35	17.71	18.16

all benchmarks is reasonably close, with exception for half splitting that performs a little worse. However, these are preliminary results and further detailed statistics are necessary to enable us to explain some apparently inconsistent results. For example, half splitting performs badly with **cubes** and **queens** benchmarks, both with incremental and without incremental copying, but, on the other hand, it is the best performing on the **nsort(10)** and **puzzle** benchmarks with incremental copying. To explain these results, we need not only to gather low level statistics, during the execution, but also understand in which manner the splitting strategy influences the scheduling of work. A postmortem visualization of the search tree might also bring some insight in to this analysis.

Although stack splitting was initially proposed for distributed memory architectures, the results show that it is equally suitable for multicore architectures. This is an interesting advantage of stack splitting since we could use it as the basis for a hybrid execution model aiming at clusters of multicores. The idea is to combine workers into teams. A team of workers might run on shared memory and use any splitting strategy to distribute work. Different teams might be assigned to different cluster nodes and can distribute work using stack splitting.

Acknowledgments

We thank the referees for their valuable comments and suggestions. This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Tech-

nology) within projects PEst (FCOMP-01-0124-FEDER-022701), LEAP (PTDC/EIA-CCO/112158/2009) and HORUS (PTDC/EIA-EIA/100897/2008).

References

1. Ali, K., Karlsson, R.: The Muse Approach to OR-Parallel Prolog. International Journal of Parallel Programming **19**(2) (1990) 129–162
2. Rocha, R., Silva, F., Santos Costa, V.: YapOr: an Or-Parallel Prolog System Based on Environment Copying. In: Portuguese Conference on Artificial Intelligence. Number 1695 in LNAI, Springer-Verlag (1999) 178–192
3. Ali, K., Karlsson, R.: Full Prolog and Scheduling OR-Parallelism in Muse. International Journal of Parallel Programming **19**(6) (1990) 445–475
4. Gupta, G., Pontelli, E.: Stack Splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In: International Conference on Logic Programming, The MIT Press (1999) 290–304
5. Pontelli, E., Villaverde, K., Guo, H.F., Gupta, G.: Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms. Journal of Parallel and Distributed Computing **66**(10) (2006) 1267–1293
6. Villaverde, K., Pontelli, E., Guo, H., Gupta, G.: PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 27–42
7. Rocha, R., Silva, F., Martins, R.: YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters. In: Portuguese Conference on Artificial Intelligence. Number 2902 in LNAI, Springer-Verlag (2003) 136–150
8. Vieira, R., Rocha, R., Silva, F.: Or-Parallel Prolog Execution on Multicores Based on Stack Splitting. In: International Workshop on Declarative Aspects and Applications of Multicore Programming, ACM Digital Library (2012)
9. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog System. Journal of Theory and Practice of Logic Programming **12**(1 & 2) (2012) 5–34
10. Villaverde, K., Pontelli, E., Guo, H., Gupta, G.: A Methodology for Order-Sensitive Execution of Non-deterministic Languages on Beowulf Platforms. In: International Euro-Par Conference. Number 2790 in LNCS, Springer-Verlag (2003) 694–703
11. Santos Costa, V., Dutra, I., Rocha, R.: Threads and Or-Parallelism Unified. Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue **10**(4–6) (2010) 417–432

Reversible Language Extensions and their Application in Debugging

Zoé Drey[1], José F. Morales[1], and Manuel V. Hermenegildo[2,1]

[1] IMDEA Software Institute, Madrid (Spain)
[2] School of Computer Science, T. U. Madrid (UPM), (Spain)

Abstract. A range of methodologies and techniques are available to guide the design and implementation of language extensions and domain-specific languages. A simple yet powerful technique is based on source-to-source transformations interleaved across the compilation passes of a base language. Despite being a successful approach, it has the main drawback that the input source code is lost in the process. When considering the whole workflow of program development (warning and error reporting, debugging, or even program analysis), program translations are no more powerful than a glorified macro language. In this paper, we propose an augmented approach to language extensions for Prolog, where symbolic annotations are included in the target program. These annotations allow selectively reversing the translated code. We illustrate the approach by showing that coupling it with minimal extensions to a generic Prolog debugger allows us to provide users with a familiar, source-level view during the debugging of programs which use a variety of language extensions, such as functional notation, DCGs, or CLP{Q,R}.

Keywords: language extensions, debuggers, logic programming, constraint programming

1 Introduction

One of the key decisions when specifying a problem or writing a program to solve it is choosing the right language. Even when using recent high-level and multi-paradigm languages, the programmer often still needs precise, domain-specific vocabulary, notations, and abstractions which are usually not readily available. These needs are the main motivation behind the development of domain-specific languages, which enable domain experts to express their solutions in terms of the most appropriate constructs.

However, designing a new language can be an intimidating task. A range of methodologies and tools have been developed over the years in order to simplify this process, from compiler-compilers to visual environments [12]. A simple, yet powerful technique for the implementation of domain-specific languages is based on source-to-source transformations. Although in this process the source and target language can be completely different, it is frequent to be just interested in some *idiomatic extensions*, i.e., adding domain specific features to a host language while preserving the availability of most of the facilities of this language. Examples of such extensions are adding functional

notation to a language that does not support it, adding a special notation for grammars (such as Definite Clause Grammars (DCGs) [15]), etc. Such transformations have been proposed in the context of object-oriented programming (*e.g.,*for Java, [14]), functional programming (*e.g.,*for Haskell, [9]), or logic programming (the `term_expansion` facility in most Prologs, or the extended mechanisms of [2,8]) In this approach, the language implementations provide a collection of *hooks* that allow the programmer to extend the compiler and implement both syntactic and semantic variations.

An important practical aspect is that, in addition to appropriate notation, the programmer also needs environments that help during program development. In particular, basic tools such as editors, analyzers, and, specially, debuggers are fundamental to productivity. However, in contrast to the significant attention given to mechanisms and tools for defining language extensions, comparatively few approaches have been proposed for the efficient construction of such development environments for domain-specific languages. In some cases ad-hoc editors, debuggers, analyzers, etc. have been developed from scratch. However, this approach is time consuming, error prone, hard to maintain, and usually not scalable to a variety of language extensions.

A more attractive alternative, at least conceptually, is to reuse the tools available for the target language, such as its debuggers or analyzers. This can in principle save much implementation effort, in the same way in which the source-to-source approach leverages the implementation of the target language to support the domain-specific extensions. However, the downside of this approach is that these tools will obviously communicate with the programmer in terms of the target language. Since a good part of the syntactic structure of the input source code is typically lost in the transformation process, these messages and debugger steps in terms of the target language are often not easy to relate with the source level and then the target language tools are not really useful for their intended purposes. For example, a debugging trace may display auxiliary calls, temporary variables, and obscure data encodings, with no trivial relation with the control or data domain at the source level. Much of that information is not only hard to read, but in most cases it should be invisible to the programmer or domain expert, who should not be forced to understand how the language at the source level is embedded in the supporting language.

In this paper, we propose a method for recovering *symbolically* the source of particular translations (that is, *reversing* them and providing an *unexpanded* view when required) in order to make target language level development tools useful in the presence of language extensions. Our solution is presented in the context of Ciao [8], which uses a powerful language extension mechanism for supporting several paradigms and (sub-)languages. We augment this extension mechanism with support for symbolic annotations that enable the recovery of the source code information at the target level. As an example application, we use these annotations to parameterize the Ciao interactive debugger, so that it displays domain-specific information, instead of plain Prolog goals. Our approach requires only very small modifications in the debugger and the compiler, which can still handle other language extensions in the usual way.

The paper is organized as follows: 2 presents a concrete extension mechanism and illustrates the limitations of the traditional translation approach in our context. 3 presents our approach to unexpansion, and guidelines for instrumenting language ex-

tensions so that the intervening translations can be reversed as needed into their input source code. 4 presents the application of the approach to the case of debuggers. Finally, 5 presents related work and 6 concludes and suggests some future work.

2 Language extensions and their limitations

We present a concrete language extension mechanism based on translations (the one implemented in the Ciao language) and then illustrate the limitations of the traditional translation-based extension approach in our context. In Ciao [8], language extensions are implemented through *packages* [2], which encapsulate syntactic extensions for the input language, translation rules for code generation to support new semantics, and the necessary run-time code. Packages are separated into compile-time and run-time parts. The compile-time parts (termed *compilation modules*) are only invoked during compilation, and are not included in executables, since they are not necessary during execution. On the other hand, the run-time parts are only required for execution and are consequently included in executables. This phase distinction has a number of practical advantages, including obviously the reduction of executable sizes.

More formally, let us assume that an extension for some language denoted as $\mathcal{L}_e$ is defined by the package PkgMod_e, and that the compiler passes include calls to a generic expansion mechanism $[\![\mathit{expand}]\!]$, which takes a package, an input program in the source language, and generates a program in the target language $\mathcal{L}$. That is, given $[\![\mathit{expand}]\!]_e = [\![\mathit{expand}]\!](\mathit{PkgMod}_e)$, for a program $P_e \in \mathcal{L}_e$ we can obtain the expanded version $[\![\mathit{expand}]\!]_e(P_e) = P \in \mathcal{L}$. Note that in practice, Ciao contains finely grained translation hooks, which allow a better integration with the module system and the composition of translations [13]. This level of detail is not necessary for the scope of this paper, and thus, for the sake of simplicity, the expansion will work on whole programs at a time.

Functional notation. We illustrate the translation process in Ciao with an example from the *functional notation* package [3]. This package extends the language with *functional*-like syntax for relations. Informally, this extension allows including terms with predicate symbols as part of data terms, while interpreting them as predicate calls *with an implicit last argument*. It also allows defining clauses in functional style where the last argument is separated by a `:=` symbol (as well as other functionalities, such as expanding goals in the last argument after the body). The translation can be abstractly specified as a collection of rewrite rules such as:

$$\begin{array}{ll} \text{(Clauses)} & \mathbf{tr}[\![\ p(\bar{a})\ \texttt{:=}\ C\ \texttt{:-}\ B\ ]\!] = (p'(\bar{v}, T)\ \texttt{:-}\ \bar{v} = \bar{a}, B, T = C) \\ \text{(Calls)} & \mathbf{tr}[\![\ q(\ldots p(\bar{a}) \ldots)\ ]\!] = (p'(\bar{a}, T), q(\ldots T \ldots)) \end{array}$$

The first rule describes the meaning of a clause in functional notation, where p' is the predicate in plain syntax corresponding to the definition of p in functional notation (i.e., using `:=`). The second rule must be applied using a leftmost-innermost strategy for every p function symbol that appears in the goal q, where T is a new variable (skipping higher-order terms). If SLD resolution is used, the evaluation order corresponds to eager, call-by-value evaluation (but lazy evaluation is possible and shown in [3]). We refer to the actual implementation later in this section.

Source code (functional notation)

```
f(X) := X < 42 ?
          (k(l(m(X))) * 3)
        | 1000.
k(X) := X + 1.
l(X) := X - 2.
m(X) := X.
```

Target code (plain Prolog)

```
f(X,Res) :- X < 42, !,
   m(X, M), l(X, L), k(X, K),
   T is K * 3,
   T = Res.
f(X,1000).
k(X,Res) :- Res is X+1.
l(X,Res) :- Res is X-2.
m(X,X).
```

Fig. 1: Example translation for functional notation.

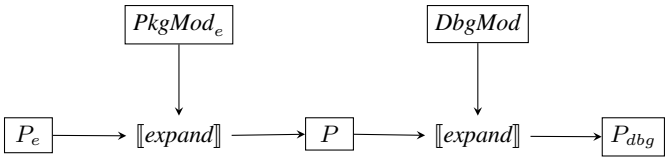


Fig. 2: The translation process and application of the standard debugger.

Example 1. In 1 we show an example program that defines a predicate `f/2` in functional notation and its translation into plain Prolog code. Its body contains nested calls to `k/2`, `l/2`, `m/2`, and also syntactic sugar for a conditional (if-then-else) construct (using the syntax: *CondGoal* ? *ThenExpr* | *ElseExpr*).

2.0.1 Forgetful Translations and Loss of Symbolic Information. Both the standard compilation and the translations for language extensions are typically focused on implementing some precise semantics during execution. That is, the correctness of the translation guarantees that for all programs $P_e \in \mathcal{L}_e$, the expected semantics $[\![exec]\!]_e$ for that language can be described in terms of a program $P \in \mathcal{L}$ and its corresponding execution mechanism $[\![exec]\!]$. That is, for all $P_e \in \mathcal{L}_e$ there exists a $P = [\![expand]\!]_e(P_e)$ so that $[\![exec]\!]_e(P_e) = [\![exec]\!](P)$.

Most of the time, symbolic information at the source level is lost, since it is not necessary at run time. In particular, such information removal and loss of structure is necessary to perform important program optimizations (*e.g.,*assigning some variables to registers without needing to keep the symbolic name, its relation to other variables in the same scope, etc.). When programs are not necessarily executed, but manipulated at a symbolic level, the translation-based approach is no longer valid on its own. For example, assume a simple *debugger* that interprets the source and allows the user to inspect variable values at each program point interactively. In this case the translation, as a program transformation, must preserve not only the input/output behaviour but also some other *observable* features (such as line numbers or variable names).

In order to explore the particular case of debuggers more closely, 2 illustrates the translation process of a source program, using a compilation module $PkgMod_e$ contain-

ing the translation rules for extension e. If the developer asks the Ciao interpreter to debug this program, further instrumentation is applied that is also defined in part as a language extension, *DbgMod* in 2; this instrumentation customizes the code by encapsulating it into a predicate that specifies whether a part of the code is *spy-able* or not. The following example illustrates in a concrete case the limitations of this process.

Example 2 (Interactive debugging). Consider the code and transformation of 1. If the target-level debugger is used without any other provision, following the process of 2, debugging a call to `f(3,T)` amounts to debugging its translation, as illustrated in the trace of 3 (the exit calls are omitted in order to save space). The problem of this trace is twofold: first, the interactive debugging does not make explicit the actual source-level predicate that is currently being tested. Second, understanding the trace forces the developer to make the mental effort of analyzing the debugged data and mapping it back to the source code. This effort increases if the source code contains operators that do not exist on the target (Prolog) side. The first case can be easily overcome when operator definitions are shared, *e.g.,*using a graphical editor and catching the operator with the line number and the occurence number of the call. However, the second case implies remembering the mapping between the source and the target operator. Furthermore, things get even more tedious and intricate when one instruction in the source language is translated into a composition of goals.

3 Building reversible extensions

In this section we provide an informal definition of *unexpansion* with respect to a language extension. We then present guidelines in order to instrument a compilation module for such a language extension. The purpose of this instrumentation is to drive the process of reconstructing a program in terms of the language extension (or *source* language) in which the program is written. Through this mechanism, a language extension can be made *reversible*. To illustrate our objective, we apply the guidelines and parameterize one of the translation rules used in the functional notation extension.

3.1 A correspondence between expansion, unexpansion, and observers

We use the term *unexpansion* to designate the inverse of the expansion $[\![expand]\!]_e$, that is, the recovering of the original P_e source program from P. Unfortunately, this inverse is rarely a one-to-one mapping. For example, `f(3,T)` in $\mathcal{L}$ corresponds to both

```
2 2 Call: f(3,_6378) ?
3 3 Call: <(3,42) ?
4 3 Call: m(3,_6658) ?
5 3 Call: l(3,_6663) ?
6 4 Call: is(_6663,3-2) ?
...
9 3 Call: is(_6673,2*3) ?
10 3 Call: =(_6378,6) ?
```

Fig. 3: Excerpt of the display of the interactive debugger.

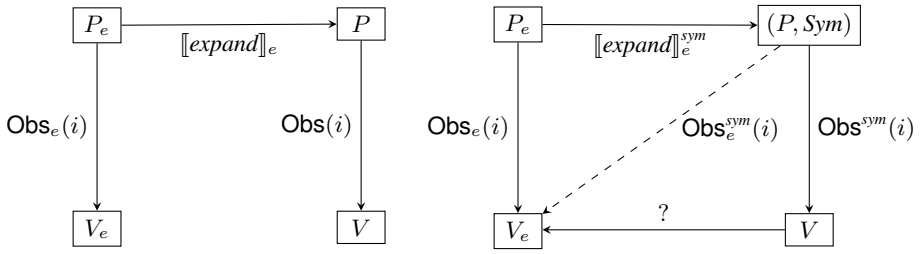


Fig. 4: Observation problem at the source level (left); Observation using symbolic information (right).

`T=f(3)` and `f(3,T)` (with `f/1` using functional notation). For another example, a clause can either be translated in one or many clauses, as depicted in Figure 1 for `f` in functional notation.

Not existing a unique solution can be confusing for the user and impractical for automatic transformations. However, the most important use of unexpansion in our context is to observe the behavior of only certain program aspects at the source language level. In this case, unexpansion seems more treatable. For that purpose we define the term *observer* accordingly: an *observer* is an interface that provides some specific source-level information about a particular program. The observer can be either static or dynamic. Specifically, we can consider as observers monitors (*e.g.,*interactive debuggers, tracers, and profilers) for dynamic observation, and verifiers (*e.g.,*static analyzers and model checkers) for static observation. Thus, a source-level view may correspond to the current instruction being invoked in an interactive debugger, or to a trace of the memory state, in a tracer, or perhaps the dependencies between the program variables, in a static analyzer, all of them represented in terms of the source language abstractions.

The correspondance between expansion and unexpansion, in the context of an observer, is sketched in Figure 4. We assume that we have observers $\mathsf{Obs}_e(i)$ and $\mathsf{Obs}(i)$ for the source and target languages, respectively. We denote by i some particular observable aspect and by V the aspect (*e.g.,*"line numbers" and an integer). On the left diagram we depict the impossibility of getting information at the $\mathcal{L}_e$ level in general. To provide the programmer with source-level observers, our approach relies on extending the expansion ($[\![expand]\!]_e^{sym}$) with additional symbolic information (which can be significantly smaller than the sources). Then, observers $\mathsf{Obs}^{sym}(i)$ can retrieve V (*e.g.,*a single number encoding the row and columns) and map it back to V_e (*e.g.,*the row and columns). This composition provides an effective $\mathsf{Obs}_e^{sym}(i)$.

We now propose guidelines for easily instrumenting the translation module of a language extension, in such a way that observers can be parameterized with respect to this instrumentation.

3.2 Instrumentation of a compilation module

Instrumenting a compilation module involves annotating its translation rules with source code information that can then be used by an observer (*i.e.,*the debugger in

our application example). We illustrate the instrumentation process on the functional extension example.

3.2.1 Guidelines. The first step in making a language extension reversible is to determine which parts of the source code need to be kept available in the expansion process. The second step is to determine how and where to propagate this information, so that it can be accessed whenever the developer requires observation during program execution. The third step is to determine the representation of the observable data.

Event and data analysis. What events do we want to observe? What do we want to observe about them? These selections should be useful for following the control flow and state changes during program execution. For example, in a λ-calculus-like language, the definition and the application of a function are two of the key elements to follow in order to debug a program [16]. As another example, in a goal involving expressions in functional notation, the debugger must be aware of which positions correspond to data terms and which positions to predicate calls.

Decomposition. How is a source statement decomposed into target code? The answer to this question implies in part how the data that we want to observe should be propagated. For example, while the generic debugger may step through a number of target-level statements, a source-specific debugger may have to consider a single source statement as corresponding to all those steps. This applies for example in the conditional statement `C ? A | B` of the functional notation, where `A` is translated into an (at least) two-goal target code segment.

Representation. How should the data to be observed be represented? In a purely syntactic extension, data always represents elements of the concrete syntax. Nevertheless, it is interesting to consider this question when displaying the runtime context, such as the state of the memory, for semantic extensions.

For example, in a CLP{Q,R} extension, variables are bound at run-time to complex terms attached to attributed variables which reflect the internal, low-level representation of the constraint store, while what the programmer would like to see is a symbolic representation of the constraints among the variables in the source constraint language.

3.2.2 Instrumentation in action. To instrument the translation rules we propose to annotate the target parameter of each rule (i.e., the argument in which the code generated by the translation is returned). This annotation (which we call the *meta-annotation*) is defined as a macro which provides the symbolic information to drive the process of recovering source code data within the observer. It may contain any data written in a prolog syntax, enabling to recover some source level information.

For example, such annotation could be a list of variables and a function enabling to recover their value in the source level notation from the target context (its environment and store), or a single string to be displayed at the observer's output at run time.

We currently distinguish two types of meta-annotations: the `$clause_info` annotation, which is wrapped around target clauses, and the `$goal_info` meta-annotation, which is wrapped around target goals. The purpose of each of these meta-annotations is to gather symbolic information to recover a source-level statement or a source-level call, respectively. Additionally, this distinction enables to handle clauses and goals properly, in particular to retrieve their location in source modules.

A meta-annotation takes two arguments: the first argument is the wrapped element (i.e., the original clause or goal(s) generated by the transformation), and the second one provides symbolic information enabling to recover an "observable" representation of the wrapped element, according to what the extension designer wants the programmer to observe. We illustrate this annotation process with Example 3.

Example 3. Let us consider the translation rule for clause declarations in the functional notation package. This rule, named `defunc`, translates such clause declarations into a set of clauses:

```
defunc((FuncHead := FuncValOpts), Clauses) :-
   FuncValOpts = (FuncVal1 | FuncValR), !,
   Clauses = [Clause1 | ClauseR],
   defunc((FuncHead := FuncVal1), Clause1),
(1)
   defunc((FuncHead := FuncValR), ClauseR).
(2)
```

The `FuncHead` part on the left corresponds to a predicate declaration; the `FuncValOpts` part on the right corresponds to goal invocations (this results from the data analysis guideline). Notice that the declaration is decomposed into many goals (marked *(1)* and *(2)*) if the `|` operator appears inside its right part. Therefore, the translation needs to be adapted slightly, in order to indicate to the debugger that the declaration is to be treated as a single one. As illustrated in Example 4 below, the resulting adaptation amounts to creating an intermediate predicate (`defunc_rec`, not really necessary in this simple case), and to annotating the `defunc` rule (this results from the decomposition guideline). Note that the `$clause_info` wrapper effectively groups all the clauses into which the definition is expanded, and this can be detected by the observer which will then treat it as a single clause.

The symbolic information attached to the annotation is represented by the contents of variable `SI`. This variable is handled by an observer, according to the nature of the program view it aims to provide. For example, line numbers, variables or function names can be attached to it. It can even be left as a free variable, in cases where the observer can automatically retrieve the information.

This approach based on meta-information enables us to envision a range of program views, from simple syntax recovery to high-level representation of analysis results: annotations can be enriched with source-specific procedures to handle various representations of the target program, enabling different instantiations of the meta-annotation variable. They can even hold procedures that perform advanced computations parameterized with the symbolic information (*e.g.,*counting the number of times a function is invoked).

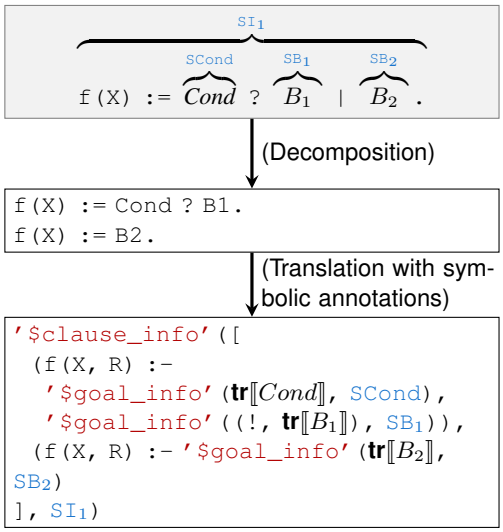


Fig. 5: Instrumented translation of a clause in functional notation.

Example 4. The instrumentation of the translation rule for declarations in functional notation writes as follows:

```
defunc((FuncHead := FuncValOpts), $clause_info(Clauses, SI)) :-
   defunc_rec((FuncHead := FuncValOpts), Clauses),
   SI = (FuncHead := FuncValOpts).

defunc_rec((FuncHead := FuncValOpts), Clauses) :-
         FuncValOpts = (FuncVal1 | FuncValR), !,
   Clauses = [Clause1 | ClauseR],
   defunc_rec((FuncHead := FuncVal1), Clause1),
   defunc_rec((FuncHead := FuncValR), ClauseR).
```

The same instrumentation method applies to goals, as outlined in the schema of Figure 5, which depicts a declaration of the form `f(X) :=` $Cond$ `?` B_1 `|` B_2. In this figure, the variable names Sx correspond to symbolic information for some program elements (like goals or clauses), and the expressions $\mathbf{tr}[\![x]\!]$ correspond to a translation of the term x. To avoid the overloading of the compilation module with annotations, symbolic information can be stored in a specific table.

4 Application to the interactive debugger

We now illustrate the use of a reversible language extension to parameterize the generic interactive debugger of Ciao. We describe the modifications performed on the compiler and on the debugger, and show the resulting source-level trace for our initial example of Figure 1.

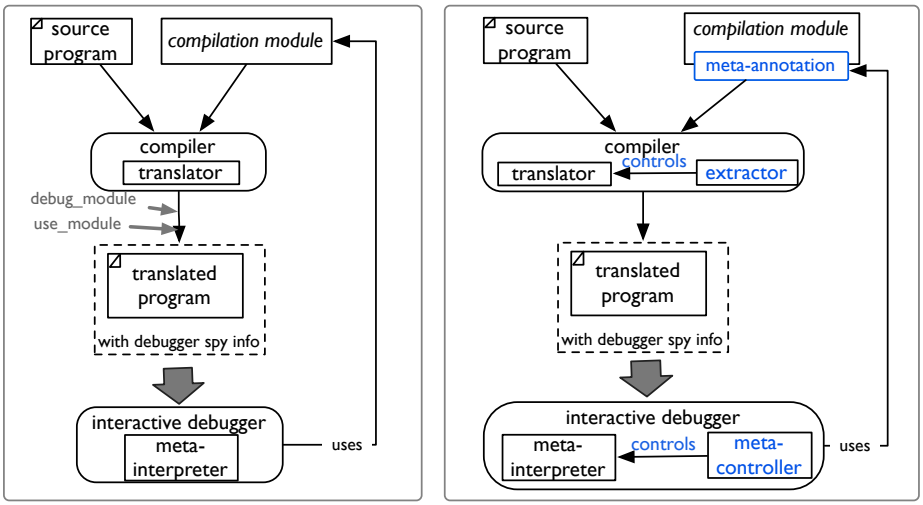


Fig. 6: Implementation: original (left) vs. customized (right) infrastructure.

4.1 Implementation details

The overall process of making program behavior observable at the source level through a debugger and reversible expansion is depicted in Figure 6.

The compiler is responsible for applying both the debugger compilation module and the source language compilation module. Prior to applying the translation rules, it extracts the elements corresponding to sentences, clauses, and goals. During this step, information to locate the source program instructions are saved, such as the module name, the line numbers for sentences, and the name of the goal being called. Then, sentences, clauses, and goals are translated according to the specifications of the corresponding compilation module. To enable the handling of the `term_info` meta-annotations in Ciao, the translation step (represented by the `translator` box in Figure 6) of the compiler needs to be customized. This is done by performing an extraction step (represented by the `extractor` box in Figure 6, right part) that modifies the translation process when a meta-annotation is encountered.

In the case of the debugger, the required symbolic information corresponds to a source node (*e.g.*,`k(X) := X + 1` as in Figure 1). As a result, the extraction process consists solely of storing each source node (either a clause or a goal) before its expansion.

Once the source-level information is extracted and mapped to the appropriate target term (or composition of target terms, *cf.* the guidelines in Section 3), it is interpreted by the debugger. To step through the source code instead of the target code, the debugger is equipped with a `meta-controller`, which checks the presence of a meta-information call at the level of the translated program, and displays a trace step accordingly. In particular, it is responsible for locating the name of the target goal in the source nodes corresponding to this goal. Since the compiler provides the source code information as a Prolog term, this localization is straightforward. When a goal invoked in

the debugger has not been annotated (with `$goal_info`), the meta-controller looks into the last `$clause_info` meta-annotation, and looks for the name of this goal inside this meta-annotation. Otherwise, the standard, expanded debug information is displayed.

4.2 Source-level tracing: the functional example revisited

With this instrumentation, Example 1 is now debugged in source code terms, as illustrated in Figure 7. Note that the debugger now displays the complete declaration (see second line) defining `f`, instead of a single part of a clause (see the second line in Example 1). When a function evaluation returns a value (which is the case of all the functions `f/1`, `k/1`, `l/1`, `m/1`), intermediate unifications are performed by the generic debugger. When the debugger is instrumented with a meta controller (*i.e.,*the handler of meta-annotations), these unification steps are ignored (skipped over), since they have no representation in the original source code.

5 Related Work

There exist frameworks and generative approaches that facilitate the development of DSL tools for programming, including debuggers [6,19]. For example, the Eclipse Integrated Development Environment [6], provides an API and an underlying framework that can greatly help in the development of a debugger [5]. Emacs is another example of such environments, with facilities in the same line as Eclipse. However, these tools are large and have a significant learning curve, and, more importantly, their facilities are centered more around the graphical navigation of the source code and interfacing with a command-line debugger, while the focus of our work is on bridging syntactic or semantic aspects between two sides of a translation, within such a command-line debugger. In that sense our work is complementary to (and in practice combines well with) the facilities in Eclipse, Emacs, and related environments. Generative approaches have been suggested (*e.g.,*based on aspect weaving into the language grammar [21]) in order to reduce developer burden when using intricate APIs.

```
2 2 Call: ex0:f(3,_6371) ?
3 3 Call: f(3) := 3 < 42 ? k(l(m(3)))*3 | 1000 ?
4 4 Call: f(3) := 3 < 42 ? k(l(m(3)))*3 | 1000 ?
5 5 Call: m(3) := 3 ?
6 4 Call: f(3) := 3 < 42 ? (k(l(m(3)))*3 | 1000 ?
7 5 Call: l(3) := 3 - 2 ?
8 4 Call: f(3) := 3 < 42 ? k(l(m(3)))*3 | 1000 ?
9 5 Call: k(1) := 1 + 1 ?
10 3 Call: f(3):= 3 < 42 ? k(l(m(3))) * 3 | 1000 ?
2 2 Exit: ex0:f(3,12) ?
```

Fig. 7: An excerpt of the debugger trace, customized with source information.

However, none of these approaches provide a methodology for developing reliable and maintainable debuggers. As a result, the development of debuggers has remained difficult, inciting DSL tool developers to implement ad-hoc solutions, through extension-specific modifications and adaptations of the debugger code. For example, SWI-Prolog includes a graphical debugger for Prolog with built-in support for DCGs and Logtalk programs [20]. As mentioned in the introduction, this approach results in useful debuggers but which are specific to concrete extensions. As a result, they have to be modified again for other transformations.

Our objective has been to develop a more general approach, which we have illustrated by applying the same methodology to several extensions including functional notation, DCGs, and CLP{Q,R}.

Lindeman *et al.* [11] have proposed recently a declarative approach to defining debuggers. To this end, they use SDF [18], a rewriting system, to instrument the abstract syntax tree with debugging annotations. However, it does not seem obvious that their approach could be applied to other observer tools. Indeed, instrumentation is achieved by providing debugger-specific information, in the form of events. In contrast, our instrumentation process makes it possible to easily add and handle different kinds of meta-information.

Unexpansion and decompilation only differ in the hypothesis used in decompilation: that the original source code may not be available. It is interesting however to compare to existing related decompilation approaches. Bowen [1] proposes a compilation process from Prolog to object code which makes it possible to define decompilation as an inverse call to compilation, provided some reordering of calls is performed. Gomez *et al.* [7] also propose a decompilation process for Java based on partial evaluation. However, these approaches have not been designed to be applicable to a large class of different language extensions. More generally, while it is in theory possible (although predictably hard with current technology) to implement fully reversible transformations, this approach runs into the problem that such inversions are non-deterministic in general, in the sense that a given target code can be generated from multiple source texts. Presenting the programmer with a different code that what is in the source program could be even more confusing that debugging the target code directly.

More similar to our solution is the approach of Tratt [17], which also targets language extensions, and where source information is injected into the abstract syntax tree of the source program. This information is exploited to report errors in terms of the language extension. However, they only discuss how to inject such information in the syntax tree, and do not explain how to use this information when building or adapting tools.

The macro-expansion passing style [4] approach makes it possible to easily implement observers. Our approach differs from this one in the reliance on the existing generic debugger (Ciao's in our examples), and concentrates instead on what changes are required in the debugger and the extension framework in order to handle meta-information for unexpansion in a way that is independent from the concrete language extension.

As a conclusion, we believe that our process proposal could be extended to other Prologs, as the meta-annotations enable to hold symbolic information that is made available in most Prolog compilers, e.g., line numbers or variable names.

6 Conclusion and future work

We have presented a generic approach that enables a debugger for a target language to display trace information in terms of the language extension in which a source program is written, using the Ciao debugger as an example. The proposed approach is based on an extension of the usual mechanisms for term expansion, and in particular of their modular implementation in Ciao through *packages*. Specifically, we define a methodology for making relevant parts of the source text and other characteristics at the target level by enriching the translation rules. We have shown that the compiler and the debugger require only small adaptations in order to take this mechanism into account and that these adaptations are generic in the sense that while the transformation rules are of course specific to the extension, the compiler and debugger themselves do not require further modification, for what is arguably a usefully large class of extensions. In particular, in the paper we have illustrated this approach by applying it on the functional notation. In the system, we have successfully applied it also to the DCG and CLP{Q,R} constraint packages.

In future work, we plan to extend the flexibility of the approach by enriching the annotations, and being able to provide different annotations for different purposes. Also, we feel that this initial work on augmenting the language extension mechanism already provides us gives with the basis for adapting the Ciao pre-processor so that for example errors, warnings, and other reports are made in terms of the source, domain-specific language, for different extensions, without requiring further modification of the pre-processor itself. The same would apply of course to the auto-documenter.

Finally, we could leverage Kishon *et al.*'s framework [10] to check the soundness of our approach with regard to the intended semantics of a language extension. Doing so would also enable to show the equivalence between the behavior of an ad-hoc source level debugger and our customization of the target level debugger.

Acknowledgments

The research leading to these results has received funding from the Madrid Regional Government under CM project P2009/TIC/1465 (PROMETIDOS), and the Spanish Ministry of Economy and Competitiveness under project TIN-2008-05624 *DOVES*.

References

1. J. P. Bowen. From programs to object code and back again using logic programming: Compilation and decompilation. *Journal Of Software Maintenance Research And Practice*, 5(4):205–234, 1993.
2. D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

3. A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *The 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, pages 142–162, Fuji Susono (Japan), April 2006.
4. R. K. Dybvig, D. P. Friedman, and C. T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, 1988.
5. Eclipse. How to write an Eclipse debugger. `http://www.eclipse.org/articles/Article-Debugger/how-to.html`.
6. ECRC. *Eclipse User's Guide*. European Computer Research Center, 1993.
7. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51(10):1409–1427, October 2009.
8. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.
9. P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.
10. A. Kishon, P. Hudak, and C. Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *PLDI*, pages 338–352, 1991.
11. R. T. Lindeman, L. C. Kats, and E. Visser. Declaratively defining domain-specific language debuggers. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 127–136, New York, NY, USA, 2011. ACM.
12. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.
13. J. F. Morales, M. V. Hermenegildo, and R. Haemmerlé. Modular Extensions for Modular (Logic) Languages. In *21th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11)*, Odense, Denmark, July 2011. To appear.
14. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction*, pages 138–152, 2003.
15. F. Pereira and D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.
16. A. P. Tolmach and A. W. Appel. A debugger for standard ml. *J. Funct. Program.*, 5(2):155–200, 1995.
17. L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):31:1–31:40, Oct. 2008.
18. M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. In *Compiler Construction*, pages 365–370, 2001.
19. M. G. J. van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju. Tide: A generic debugging framework — tool demonstration —. *Electron. Notes Theor. Comput. Sci.*, 141(4):161–165, Dec. 2005.
20. J. Wielemaker. SWI-prolog — source-level debugger. `http://www.swi-prolog.org/gtrace.html`.
21. H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1370–1374, New York, NY, USA, 2005. ACM.

Parallel Local Search: Experiments with a PGAS-based programming model

Rui Machado[1,2], Salvador Abreu[2], and Daniel Diaz[3]

[1] Fraunhofer ITWM, Kaiserslautern, Germany
rui.machado@itwm.fhg.de
[2] Universidade de Évora and CENTRIA, Portugal
spa@di.uevora.pt
[3] University of Paris 1-Sorbonne, France
Daniel.Diaz@univ-paris1.fr

Abstract. Local search is a successful approach for solving combinatorial optimization and constraint satisfaction problems. With the progressing move toward multi and many-core systems, GPUs and the quest for Exascale systems, parallelism has become mainstream as the number of cores continues to increase. New programming models are required and need to be better understood as well as data structures and algorithms. Such is the case for local search algorithms when run on hundreds or thousands of processing units. In this paper, we discuss some experiments we have been doing with Adaptive Search and present a new parallel version of it based on GPI, a recent API and programming model for the development of scalable parallel applications. Our experiments on different problems show interesting speedups and, more importantly, a deeper interpretation of the parallelization of Local Search methods.

Keywords: Parallel Local Search, GPI, Adaptive Search, Constraint Programming

1 Introduction

Systematic and complete search algorithms impose a limitation on the problem size they are able to solve due to the exponential increase in processing time and memory requirements. For this reason, heuristics-based search algorithms are used (and necessary) for larger problem sizes. Instead of exploring the complete search space, heuristics are used to guide the search to portions of the search space where solutions might be found. Local Search and Meta-heuristics are an interesting paradigm for combinatorial search and have been shown very effective for solving real-life problems [9,8]. But despite the effectiveness of local search methods, for really large problem instances, the running time required might still be substantial. One way to cope with this problem is by introducing parallelism.

The current trend we are facing is an inevitable paradigm shift towards multi-core technologies where parallelism is now omnipresent. In recent systems parallelism spreads over several systems levels and heterogeneity is growing on the node as well as on the chip level. Data must be maintained across a hierarchy of memory levels and

most applications and algorithms are not yet ready to take full advantage of available capabilities. There is a demand for programming models with a flexible threads model and asynchronous communication to cope with this gap.

PGAS (Partitioned Global Address Space) programming models have been discussed as an alternative to MPI [12] for some time. The PGAS approach offers the developer an abstract shared address space which simplifies the programming task and at the same time facilitates data-locality, thread-based programming and asynchronous communication. GPI is a PGAS API that follows this philosophy and delivers the full performance of RDMA-enabled[4] networks directly to the application without interrupting the CPU.

In this paper we aim at bringing together both the need for parallelism to solve large problem instances with Local Search and its availability in current systems. We implemented a new parallel version of the Adaptive Search algorithm based on GPI that goes beyond the simple independent multiple-walk. Our new design shows interesting speedup gains on benchmarks with scalability problems and more importantly, a deeper interpretation on the parallelization of Adaptive Search in particular and Local Search methods in general, based on some characteristics of the benchmarks.

The rest of the paper is organized as follows: in section 2 we present GPI and its programming model, hightlighting some its major features. Section 3 provides some background on the Adaptive Search algorithm and section 4 focuses on its parallelization. In section 5, we detail our parallelization strategy based on GPI and in section 6 we show the obtained results and compare it to the previous implementation. Section 7 examines and interprets our experimental findings, correlating them with the characteristics of the problems. Finally, section 8 presents a short conclusion and perspectives of future work.

2 GPI

GPI (Global address space Programming Interface) is a PGAS API for parallel applications running on clusters. The thin communication layer delivers the full performance of RDMA-enabled networks directly to the application without interrupting the CPU. Fig. 1 depicts the architecture of GPI.

The local memory is the internal memory available only to the node and allocated through typical allocators (e.g. malloc). This memory cannot be accessed by other nodes. The global memory is the partitioned global shared memory available to other nodes and where data shared by all nodes should be placed. The DMA interconnect connects all nodes and is through this interconnect that GPI operations are issued. At the node level, the Manycore Threading Package (MCTP) is used to take advantage of all cores present on the system and make use of the GPI functionality and global memory. The MCTP was developed to help programmers take better advantage of new architectures and ease the development of multi-threaded applications. The MCTP is a threading package based on thread pools that abstracts the native threads of the platform.

[4] RDMA - Remote Direct Memory Access.

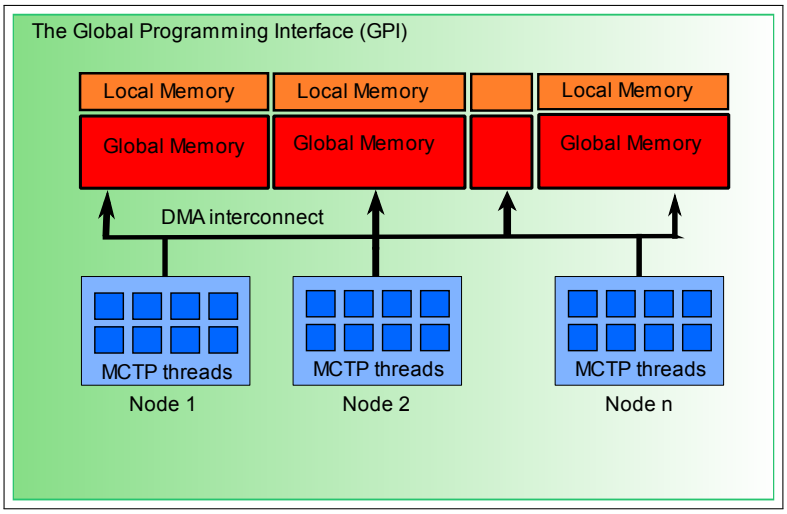


Fig. 1: GPI

GPI is constituted by a pair of components: the GPI daemon and the GPI library. The GPI daemon runs on all nodes of the cluster, waiting for requests to start applications and the library holds the functionality available for a program to use: read/write global data, passive communication, global atomic counters, collective operations. The two components are described in more detail in our previous contribution [10].[5]

The GPI core functionality can be summarized as follows:

- read and write global data
- passive communication
- send and receive messages
- commands
- global atomic counters and spinlocks
- barriers
- collective operations

In the context of this work, the most important functionality is the read/write of global data.

Two operations exist to read and write from global memory independent of whether it is a local or remote location. One important point is that those operations are one-sided that is, only the peer that issues such operation takes part in it. This is different from a two-sided scheme (message passing) where the peer that sends (*sender*) has a corresponding peer (*receiver*) that needs to issue a receive operation. Moreover, this functionality is non-blocking and completely off-loaded to the interconnect, allowing the program to continue its execution and hence take better advantage of CPU cycles. The data movement does not require any intermediate buffers and protocols to maintain

[5] GPI was previously known as Fraunhofer Virtual Machine (FVM).

those buffers. If the application needs to make sure the data was transferred (read or write), it needs to call a wait operation that blocks until the transfer is finished and asserting that the data is usable.

3 Adaptive Search

Local Search is based on the simple idea of "searching" by iteratively moving from one solution to one of its *neighbours*. The neighborhood of a solution is the set of solutions that can be obtained by applying a *move*. A *move* is a local change (hence the name Local Search).

The mechanism used to select a neighbour and thus the definition of what constitutes a neighbourhood is the main issue that differentiates between different local search methods. In general, it is problem dependent and is related to the definition of the *objective function*.

The Adaptive Search method [4] is one of many different local search methods and has proved to be very efficient in the types of problems where it was tested. It is a generic, domain-independent constraint-based local search method.

This meta-heuristic takes advantage of the structure of the problem in terms of constraints and variables and can guide the search more precisely than a single global cost function to optimize, such as for instance the number of violated constraints. The algorithm also uses an short-term adaptive memory in the spirit of Tabu Search in order to prevent stagnation in local minima and loops. This method is generic, can be applied to a large class of constraints (e.g. linear and non-linear arithmetic constraints, symbolic constraints, etc) and naturally copes with over-constrained problems.

The input of the method is a problem in CSP format, that is, a set of variables with their (finite) domains of possible values and a set of constraints over these variables. For each constraint, an "error function" needs to be defined; it will give, for each tuple of variable values, an indication of how much the constraint is violated. For instance, the error function associated with an arithmetic constraint $|X - Y| < c$, for a given constant $c \geq 0$, can be $max(0, |X - Y| - c)$.

Adaptive Search relies on iterative repair, based on variable and constraint error information, seeking to reduce the error on the worst variable so far. The basic idea is to compute the error function for each constraint, then combine for each variable the errors of all constraints in which it appears, thereby projecting constraint errors onto the relevant variables. Finally, the variable with the highest error will be taken and its value will be modified. In this second step, the well known min-conflict heuristic is used to select the value in the variable domain which is the most promising, that is, the value for which the total error in the next configuration is minimal. In order to prevent being trapped in local minima, the Adaptive Search method also includes a short-term memory mechanism to store variables to avoid (variables can be marked Tabu and "frozen" for a number of iterations). It also integrates reset transitions to escape stagnation around local minima. A (partial) reset consists in assigning fresh random values to some variables (also randomly chosen). A reset is guided by the number of variables being marked Tabu. As in any local search method, it is also possible to restart from scratch when the number of iterations reaches a given limit.

4 Parallel Adaptive Search

When parallelizing an algorithm one aims at identifying hotspots and sources of parallelism. As with most of meta-heuristics, in Adaptive Search these sources of parallelism are essentially: (1) the inner loop of the algorithm *i.e.,*computing and combining the errors of variables and selecting the variable with highest error and (2) the search space of the problem.
The problem with exploiting the inner loop of the algorithm is its granularity: it is too fine-grained. The overhead associated with synchronization and dispatching of tasks comes at a too high cost.

The other main source of parallelism is the search space (domain) of the problem itself. Theoretically, this domain could be decomposed in several disjunct partitions to be explored in parallel and without dependencies. However, in practice, several issues arise with this. Each partition is in general still too large for a sequential execution and more importantly, not the whole search space is equally valid and the exploration should avoid areas of it that lead to poor solutions. Moreover, it is hard and expensive to control and maintain the search conducted in the different partitions since a Local Search algorithm only has a local view of the search space. One example is the class of problems that have the best solutions clustered in a certain 'zone' of the search space. In this case, the algorithm should converge to that zone but in case of parallel execution avoid too much redundant work.

The Adaptive Search method has already been subject to some research on its parallel behaviour. Previous work on parallel implementations of the Adaptive Search algorithm have mostly focused on independent multiple-walks, requiring no commmunication neither shared memory between processing units.

In [5], the authors present a parallel implementation of the Adaptive Search algorithm for the Cell/BE, a heterogenous multicore architecture. The system includes 16 processors (the SPEs) where each one starts with a different random initial solution. The PPE acts as the master processor, waiting for the message of a found solution. For such number of processing units, the results were very promising, achieving for some problems linear speed-up.

Further work with Parallel Adaptive Search continued to follow the same approach with no communication between workers but more interestingly, concentrating on cluster systems with a larger number of cores.

In [2], the authors experiment and investigate the performance of a multiple independent-walk on a system with up to 256 cores. The parallelization was done with MPI and involves the introduction of a "communication step" which tests if termination was detected (a solution was found) and terminates the execution properly.

The presented performance results are relatively modest in terms of parallel efficiency and still far away for the ideal speed-up which contrasts with the results obtained at a smaller scale (ie. up to 16 cores) in previous work. This points out the need for better alternative strategies in order to better exploit large-scale parallelism.

Since that the independent multiple-walk approach still leaves space for improvement in terms of parallel efficiency and scalability for some problems, new ways to take take full advantage of parallel systems must be found.

In [1], the authors experiment with more complex strategies, where processes exchange messages resembling branch-and-bound methods where the bound is exchanged between all participants. In their work, two alternatives are attempted: exchanging the cost of the current solution of each process and the current cost plus the number of iterations needed to achieve that cost. Unfortunately, both approaches do not achieve better results than an independent multiple-walk.

5 Adaptive Search with GPI

Previous work with parallel Adaptive Search provides some groundwork to build upon and has showed that some benchmarks exhibit scalability problems when run on a large number of cores.
GPI seems, *à priori*, an interesting match to the problem of parallelization. Local search methods work with local information, trying to progress and converge to solutions in a global search space, requiring low global information. However, in a parallel setting, communication and cooperation are crucial and in this case, required to overcome the low parallel efficiency in some problems. The communication with GPI is based on one-sided primitives that might benefit the local view on a global search space, allowing threads to cooperate asynchronously. Moreover, communication is very efficient as GPI exploits the full performance of the interconnect. Hence, we continue to explore ways to further improve the parallelization of the Adaptive Search algorithm, exploiting GPI and its programming model, with the objective of getting some further benefits. But more importantly, to find mechanisms, concepts or limitations that are general.

In general, we can define the following objectives:

- further investigate and understand the behavior of parallel Adaptive Search on different problems.
- investigate the possibilites given by GPI and devise more complex mechanisms for the parallel execution of Adaptive Search, improving its performance
- identify the, possibly new, problems generated by the previous point.

The new parallel version of Adaptive Search based on GPI includes two variants which we name TDO (Termination Detection Only) and PoC (Propagation of Configuration).

The TDO variant implements the simple independent multiple-walk and serves mostly has our basis for comparison. First, with the existing MPI version, making sure that the implementation is correct and the performance is as expected. Second, to allow us to measure the improvement (if present) obtained with the more complex PoC variant. The PoC variant introduces more communication and sharing between working threads, by means of GPI primitives and threaded model.

The next sections present the two different variants in more detail.

5.1 Termination Detection Only

The variant with Termination Detection Only (TDO) is rather straightforward and implements the idea of an independent multiple-walk: all available cores execute the sequential version of the Adaptive Search algorithm.

We name this variant as Termination Detection Only since it subsumes itself to a termination detection problem *i.e.,*detecting the termination of a distributed computation. Termination Detection is itself a subject of much research and several algorithms have been and continue to be proposed([6,11,13]).

In the case of the Parallel Adaptive Search method, we are interested in detecting termination as soon as one of the participating threads has found a solution, instead of waiting for all threads to finish as some of them can potentially require too many steps in order to find a solution (it is enough to be trapped in a 'zone' of the search space with no possible solutions).

The implementation of this variant is simple as it only involves the implementation of a mechanism of triggering and detecting termination.

The GPI implementation follows a similar line of the previous work with MPI. Whenever a thread finds a solution, it triggers termination by writing to its peers that it has found a solution. Thus, the time of the parallel execution is the time taken by this fastest thread.

Other threads must detect termination. This is only possible by introducing a communication step inside the internal loop of the Adaptive Search algorithm. This is required since there is no other way for a GPI instance to react on an remote event (*i.e.,*termination) other than with communication. In this communication step, a check for termination is done on a particular memory address that is written on termination emmission as described above. The communication step introduces some overhead that needs to be minimized. Thus the communication step is only executed every k iterations.

5.2 Propagation of configuration

The experiments in previous work and with the TDO variant have found that the simple approach to parallelization, namely, the independent multiple-walk, proves itself insufficient in obtaining parallel efficiency on some problems specially when experimenting with a large number of cores. Moreover, exchanging some simple information such as the cost leads to no improvement.

Hence, we aim at communicating more and more meaningful information, introducing cooperation. By cooperation we mean mechanisms that allow threads to share information about their state and thus benefit from the collective search. Also, we want to exploit the potential and benefits of GPI and its programming model (one-sided communication, no wait for communication, global access to data, threaded model, etc.).

One of the most powerful aspects of Local Search is its simplicity. And due to this simplicity, it is hard to extract what could be considered as meaningful information to be shared and communicated. One logical candidate not yet tried is the whole current solution or configuration. Because the term *solution* is sometimes misleading, we refer to the current solution as a *configuration*. The final solution represents the solution when the algorithm stops.

The used implementation of the Adaptive Search method deals only with permutation problems and thus, a configuration is the permutation vector of the problems' variables.

Similarly to other approaches to the parallelization of local search methods which introduce cooperation, several important questions arise, namely:

1. Who does the communication?
2. When to do the communication?
3. How to do the communication?
4. What to communicate?

Answering most of these questions requires carrying out actual experiments since the best and correct answer it is not, in our opinion, foreseeable.

Our approach, which we call Propagation of Configuration (PoC), aims at answering these questions and give a better understanding of how cooperation can help with increasing the scalability of Local Search in general and the Adaptive Search method in particular.

Who does the communication?

Answering the question of who does the communication involves deciding whether a single thread or all threads actually perform communication. Note that by communication we mean that, in a distributed setting, messages between nodes are exchanged. In a single node and given the GPI programming model, we can benefit from the threaded-model and shared memory. Notwithstanding the best option for this, it is clear that all threads must benefit from it.

There are potential advantages and disvantages with both options. If all threads perform communication, any shared resources must be protected by a mutual exclusion mechanism, which might suffer from high contention. Moreover, when all threads perform communication a lot more pressure on the interconnect follows, increasing the parallel overhead and with possibly a lot of redundant communication happening (the same configuration being communicated several times). But, on the other hand, there will be a rapid progress towards the best promising neighborhood, intensifying the search. Of course, this can be positive but can also become dangerous since most of threads might get trapped in a local minimum or poor quality neighborhood. A good trade-off between intensification and diversification must be achieved.

If a single master thread communicates, the effects are potentially the opposite: less intensification but also less contention, less pressure on the interconnect and less redundant work.

Preliminary tests made clear that the best option is the one with a single communicating thread since it reduces the parallel overhead. Plus, with GPI, all threads in a single node benefit immediately from the results obtained by the master thread without any exchange of messages.

When to do the communication?

The first possible answer to this question is to follow the same strategy as with the Termination Detection Only variant: introduce a communication step and perform communication every k iteration. The value of k is fundamental on how well this option might perform. With a low value (*e.g.*,$k = 10$), a strong intensification of the search is achieved but with the danger that threads might give up too soon on a promising neighborhood.

With a high value of k, we avoid that danger but less intensification will be achieved since less information will be propagated.

The other option is to not interrupt the normal flow of the algorithm for communication, letting the search progress normally and independently until a local minimum is achieved. Only at this point the configuration is propagated and possibly used. One danger however is if threads don't hit local minima that often, the propagation of configuration won't progress and some threads might never see an up-to-date configuration, achieving less intensification. A solution to this problem is to still have communication every k iteration, where threads simply keep the communication progressing but only use the propagated information when they are in trouble *i.e.,*hit a local minimum. However, this option increases the overhead by adding the extra communication step in some iterations.

In principle the second option might seem more promising as no disturbance is caused when the algorithm is progressing positively. But the forementioned danger that the propagation of configurations won't progress can have the consequence that there won't be a benefit from the communication scheme when compared to the simple TDO variant. We performed some tests on a problem with low number of local minima (Magic Square) and in fact, this is what happens.

Based on this reasoning, our chosen option to when to communicate is to have a communication step. Moreover, we still need to detect termination thus a communication step must be present, even if with a much lower influence in terms of overhead. Our PoC variant combines termination detection and the propagation of configurations in a single step that happens every k iterations and we focus on finding an optimal value for k.

How to do the communication?

With this question, we consider a single alternative. Since we aim at large scale executions, we need an efficient approach. Communication is done in a tree-based topology, in which each node only communicates with its parent and children (if any). Currently, a binary tree is used but this can be parametrized at initialization. At each communication step, the propagation of the configuration is done either up (to parent) or down (to the children) the tree. This only happens if a configuration was propagated from the children (in case of the up direction) or from the parent (down direction). The propagation of the communication behaves then like a wave, up and down the tree, with possibly different configurations being propagated at different points of the tree and contributing to some diversification.

Communication is performed by using GPI one-sided primitives. A thread posts a write operation and returns immediately to work. The configuration to be propagated will be directly written to the memory of the remote node asynchronously, without any acknowledgement of it and overlapped with the algorithm's computation. The remote node on the other hand, on its communication step, checks if a valid configuration was written to its memory, decides how to act on it and propagates its decision further.

We consider this single alternative since it gives us a good balance between intensification and diversification and because having a tree-based topology provides an efficient pattern to achieve communication scalability. The final objective is to have a

communication step with low overhead and here GPI provides us with mechanisms to do so.

What to communicate?

The Adaptive Search method (as many other Local Search methods) is very simple and includes very few elements that can be communicated.

The proposed option already mentioned before, is to communicate a full configuration. To this, we only add the cost of the configuration as it is the metric to evaluate the configuration. Plus, computing the cost everytime we communicate a configuration is a source of extra overhead specially if a problem has a large number of variables.

Still, the question remains of which configuration to communicate. In our design the best configuration *i.e.,*the configuration with better cost is communicated. At a communication step, a thread decides to propagate its own current configuration or the propagated configuration(s).

Communicating configurations can be of advantage because it includes implicitly more information about the state of the search since it, in a sense, provides a semi-exact positioning within the whole search space. As the best configurations are being propagated, other threads that are currently on poorer neighborhoods might benefit from moving to the best ones. With the stochastic behavior of Adaptive Search and enough diversification, the whole search procedure can be performed on the best neighborhoods and possibly, converge faster to good solutions.

6 Experimental results

In this section we present the obtained results using different problems.

- **all-interval**: the All Interval Series problem (prob007 in CSPLib [7]),
- **costas-array**: the Costas Array problem,
- **magic-square**: the Magic Square problem (prob019 in CSPLib).

The experiments were conducted on a cluster system where each node includes a dual Intel Xeon 5148LV ("Woodcrest") (*i.e.,*4 CPUs per node) with 8 GB of RAM. The full system is composed of 620 cores connected with Infiniband (DDR). Since we aim at large scale, we performed our experiments on the system using up to 256 cores on some problems and 512 cores on others. This difference is due to the fact that the system is largely used and is hard to get access to the full system.

Note that Adaptive Search, as many other Local Search methods, has a stochastic behavior to achieve diversity on the search. To benchmark such behavior, several executions must be done and averaged. In our experiments we ran each problem 100 times in order to obtain meaningful results.

We compare both GPI variants (TDO and PoC) with the MPI implementation from previous work, which serves as our basis for comparison.

Fig. 2 depicts the obtained results for the Costas Array problem (CAP) with n=20.

As already observed in previous work [3], the CAP shows an almost optimal scalability using an independent multiple-walk with no cooperation. We can observe that

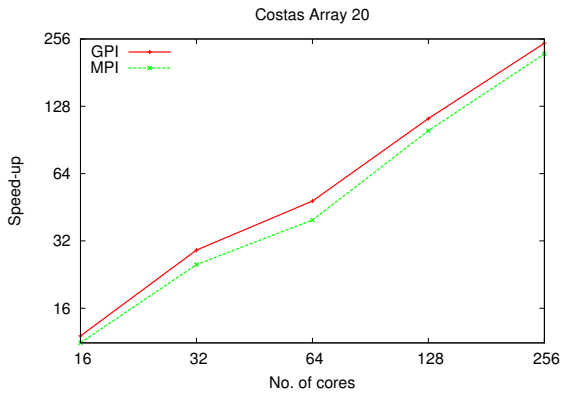


Fig. 2: Costas Array (n=20) on 256 cores (64 nodes)

our implementation obtains similar, although slight better, results. This is the expected result since both approaches (TDO and MPI) are equivalent and a confirmation that our implementation performs as expected.

Although we aspired at obtaining even better results with the PoC variant (possibly super linear) for this problem, our experiments showed that this variant performs much worse than the simple TDO variant and thus we only present the speedup obtained with GPI using the TDO variant.

The Fig. 3 depicts the obtained results for the Magic Square problem up to 512 cores.

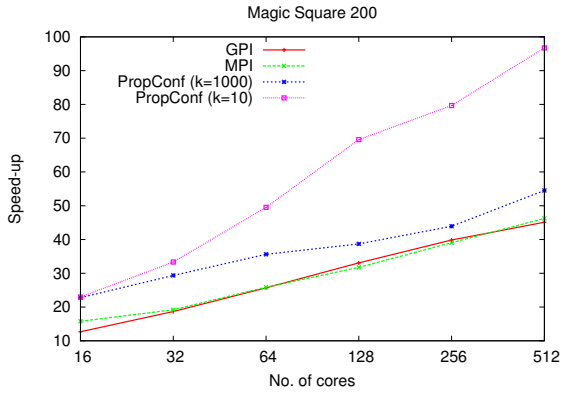


Fig. 3: Magic Square 200 on 512 cores (128 nodes)

For this problem we present the speedup obtained with the TDO and PoC variants and compare it with the MPI version. The GPI TDO variant presents again, as expected, results similar to the MPI version.

The Magic Squares benchmark is one of the problems that results in disappointing scalability when using the simple independent multiple-walk and therefore a major target for improvement with more sophisticated approaches. Indeed, for this problem, our PoC variant improves the performance and scales better as we increase the number of cores used.

We wanted to answer the question of when to do communication: as we mentioned, in our preliminary experiments it turned out that the best approach is to have a communication step every k iterations where the value of k is decisive. Surprinsingly, for this benchmark, a lower value of k (k=10 in contrast to k=1000) improves scalability by a factor of 2, achieving a speedup of 97 with 512 cores. Still a low parallel efficiency but a large improvement over the other options and variants.

The obtained results for the last problem, the All Interval series (n=400), is shown in Fig. 4.

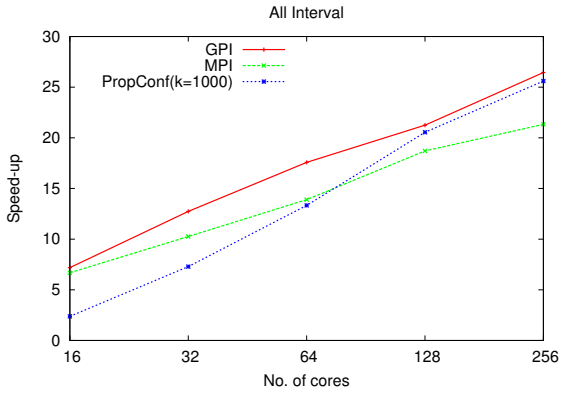


Fig. 4: All Interval 400 on 256 cores (64 nodes)

The All Interval Series benchmark is also one of the problems where good scalability was hard to reach when using a large number of cores. In Fig. 4 it is possible to observe this fact, where both the MPI and GPI TDO versions reach a modest speedup factor of 20 and 25, respectively (with 256 cores). Our PoC variant however, performs much worse than the TDO variant at a low number of cores but it improves as we increase the number of cores, hinting that this variant can be of advantage if we increase the number of cores and the problem size. In Fig. 4 we only depict the obtained results for the PoC variant with k = 1000 since, for this benchmark, it is the best value. In contrast to the Magic Squares benchmark, a lower value of k results in a much worse performance.

7 Discussion

The experimental results presented large differences in how the different problems benefit from parallelism and the implemented variants. One of our main objectives is to investigate and understand why this happens.

In order to be able to draw some conclusions on our experiments, it is important to characterize the chosen problems from different perspectives. We characterize the problems using different information such as the number of iterations and local minima. This characterization will give us a basis to better understand the problems at hand and possibly explain our results.

The Table 1 presents the obtained values for acquired information when running some instances of the previously presented problems. This information is the following:

Problem The problem instance.
Iterations The number of iterations required to find a solution.
Local Minima The number of local minima found.
Resets The number of partial resets performed (not full restart).
Same var / Iteration The number of times that existed more than one candidate variable (highest error value) to be selected.

This information allows us to better understand how does the Adaptive Search algorithm progress towards a solution, the neighborhood structure and extract further information (*e.g.,*number of local minima *per* iteration).

Problem	Iterations	Local Minima	Resets	Same var/Iteration
Magic Square 200	413900.505	25864.75	3.01	23.36
Costas 18	389932.263	204024.89	204024.89	1.00
Costas 19	3364807.772	1714299.50	1714299.50	0.99
All Interval 200	11229.220	495.27	495.27	5.97
All Interval 400	41122.406	1422.15	1422.15	9.19

Table 1: Information collected for different problems instances.

From Table 1 we can see that the different problems exhibit a quite different behavior. The Magic Square problem performs a low number of partial resets when compared to the total number of iterations or to the number of identified local minima. On the other hand, it is the problem where the number of candidate variables per iterations (Same var/Iteration) is high, meaning that at each iteration there are several possible moves towards the next configuration.

The Costas Array problem exhibits a completely different behavior. In this problem, the number of local minima identified is very large (almost every second iteration finds a local minimum) and the number of partial resets is very high, coincident with the

number of local minima *i.e.,*at each local minimum found, a partial reset is performed. Also the number of possible moves at each iteration is close to 1.

The All Interval problem is yet another kind problem. Here, the number of resets is as with the CAP equal to the number of local minima but these happen much less often. The number of possible variable choices or moves is higher than 1, meaning that some diversification could be achieved.

If we relate this characterization of problems with the obtained experimental results, some conclusions can be conjectured in order to better understand the parallelization of such algorithm or, more concretely, how much can it benefit from a communication scheme such as the one we designed.

We argue that one critical aspect is the neighborhood of a configuration or the set of possible moves, which define transitions between configurations. Since we are propagating configurations we can look at our problems at hand according to this aspect. If a problem has a dense neighborhood or, in other words, the set of possible moves at each transition is (much) larger than one, each of these moves can be explored in parallel. Thus, when a promising configuration is propagated and several moves are possible and explored in parallel, the probability that one of these moves leads to a faster path towards an optimal solution increases.

Another important aspect is the number of local minima and resets and how both relate. A problem that finds a large number of local minima before encountering an optimal solution benefits less from processing a configuration which seems promising. This configuration is heuristically promising but in reality this information is less meaningful than it should. Similarly, a problem with a high number of partial resets suffers from the same problem.

Looking back at our experimental results with the different problems, we can better understand a) the difference in scalability and b) the improvement factor brought by the PoC variant to some problems.

In the Magic Square problem, each configuration has a dense neighborhood and benefits from the parallel exploration of different moves. Thus, the PoC variant improves the performance and scalability of the algorithm. When a working thread adopts a propagated configuration, it will define its own path from that configuration and differently from one other thread that receives that same promising configuration. Moreover, this problem has a low number of local minima and resets meaning that paths from one (initial) configuration towards an optimal solution are a series of transitions from neighbor configurations.

The Costas Array Problem exhibits optimal scalability with the independent multiple-walk MPI version or with our TDO variant and this is already *per se* satisfactory. On the other hand, it performs worse with the PoC variant: propagating a configuration is only a source of parallel overhead and will limit the search allowing less diversification. A propagated configuration will allow, on average, a single move and two threads taking the same configuration results in redundant work which is also probably unfruitful since the CAP is one of the problems with a high number of local minima and reset. This also explains the good scalability using the TDO variant, where increasing the number of cores allows covering more of the total search space together with the fact that solutions for this problem are well spread over it.

Finally, the All Interval Series problem shows a mixed behavior. Similarly to the CAP, the larger number of local minima found and same number of resets point to the same problem. There is less benefit from taking a propagated configuration since its meaningfulness is low. The PoC variant only introduces unnecessary overhead and this could explain the much worse performance at a lower number of cores. On the other hand, and similarly to the Magic Square benchmark, there is more than one possible move, on average *i.e.,*some diversification can be achieved. With a large enough number of cores, the parallel overhead can be amortized by the gain obtained with this diversification. This could be the reason for the steeper curve for the PoC variant on Fig. 4. Of course, with further experiments we will be able to understand this better.

In summary, problems where configurations have a denser neighborhood benefit from a cooperation scheme such as the PoC variant where the full configuration is communicated. Contrarily, problems that follow a trajectory with a single move possible won't benefit from a communication scheme that propagates the best current configuration. Also, if a large number of local minima is found and partial resets are required in the same number, the expectation for improvement in performance is zero.

8 Conclusion

In this paper we presented our work on the parallel implementation of the Adaptive Search method using a different programming model. GPI is an API designed for high-performance and scalable parallel applications. We aimed at investigating and understanding the behavior of Adaptive Search in a parallel setting, focusing on different problems particularly those that, in previous work, showed scalability problems when targeting a large number of cores. GPI and its programming model allowed us to design a new communication and parallelization scheme which in our experimental evalution allowed a gain of a factor of 2 in terms of speedup for some problems. More importantly, it provided deeper insight and understanding on the parallelization of Local Search methods given different problems with disparate characteristics such as the neighborhood of a configuration, the number of local minima and partial resets.

In the future, we intend to examine our design and conclusions with other larger problems and experiment with complexer parallelization schemes. One possible direction is instead of using promising information (configurations, cost, statistics) directly, act on the complement of it, avoiding redundant work and cover as much as possible from the search space since this is the main source of parallelism.

One of our potential final goals is the design of a new Local Search algorithm more amenable to parallelization that builds upon these experiences.

References

1. Yves Caniou and Philippe Codognet. Communication in parallel algorithms for constraint-based local search. In *IPDPS Workshops*, pages 1961–1970, 2011.
2. Yves Caniou, Philippe Codognet, Daniel Diaz, and Salvador Abreu. Experiments in parallel constraint-based local search. In Peter Merz and Jin-Kao Hao, editors, *EvoCOP*, volume 6622 of *Lecture Notes in Computer Science*, pages 96–107. Springer, 2011.

3. Yves Caniou, Daniel Diaz, Florian Richoux, Philippe Codognet, and Salvador Abreu. Performance analysis of parallel constraint-based local search. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 337–338, New York, NY, USA, 2012. ACM.
4. P. Codognet and D. Diaz. Yet another local search method for constraint solving. *Stochastic Algorithms: Foundations and Applications*, pages 342–344, 2001.
5. Daniel Diaz, Salvador Abreu, and Philippe Codognet. Targeting the cell broadband engine for constraint-based local search. *Concurrency and Computation: Practice and Experience*, 24(6):647–660, 2012.
6. Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Inf. Process. Lett.*, 16(5):217–219, 1983.
7. Ian P. Gent and Toby Walsh. Csplib: A benchmark library for constraints. In *CP*, pages 480–481, 1999. `http://www.csplib.org`.
8. T. Gonzalez, editor. *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall / CRC, 2007.
9. T. Ibaraki, K. Nonobe, and M. Yagiura, editors. *Metaheuristics: Progress as Real Problem Solvers*. Springer Verlag, 2005.
10. Rui Machado and Carsten Lojewski. The fraunhofer virtual machine: a communication library and runtime system based on the rdma model. In *Computer Science-Research and Development*, volume 23(3), pages 125–132, 2009.
11. Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987. 10.1007/BF01782776.
12. MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009. available at: `http://www.mpi-forum.org` (Dec. 2009).
13. Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar B. Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In *PPOPP*, pages 201–212, 2011.

The Ciao CLP($\mathcal{FD}$) Library
A Modular CLP Extension for Prolog
(System Description)

Emilio Jesús Gallego Arias[1], Rémy Haemmerlé[1],
Manuel V. Hermenegildo[1,2], and José F. Morales[2]

[1] Universidad Politécnica de Madrid
[2] IMDEA Software Institute

Abstract. We present a new free library for Constraint Logic Programming over Finite Domains, included with the Ciao Prolog system. The library is entirely written in Prolog, leveraging on Ciao's module system and code transformation capabilities in order to achieve a highly modular design without compromising performance. We describe the interface, implementation, and design rationale of each modular component. The library meets several design goals: a high level of modularity, allowing the individual components to be replaced by different versions; high-efficiency, being competitive with other $\mathcal{FD}$ implementations; a glass-box approach, so the user can specify new constraints at different levels; and a Prolog implementation, in order to ease the integration with Ciao's code analysis components. The core is built upon two small libraries which implement integer ranges and closures. On top of that, a *finite domain variable* datatype is defined, taking care of constraint reexecution depending on range changes. These three libraries form what we call the $\mathcal{FD}$ *kernel* of the library. This $\mathcal{FD}$ kernel is used in turn to implement several higher-level finite domain constraints, specified using indexicals. Together with a labeling module this layer forms what we name *the $\mathcal{FD}$ solver*. A final level integrates the CLP($\mathcal{FD}$) paradigm with our $\mathcal{FD}$ solver. This is achieved using attributed variables and a compiler from the CLP($\mathcal{FD}$) language to the set of constraints provided by the solver. It should be noted that the user of the library is encouraged to work in any of those levels as seen convenient: from writing a new range module to enriching the set of $\mathcal{FD}$ constraints by writing new indexicals.

1 Introduction

Constraint Logic Programming (CLP) [1] is a natural and well understood extension of Logic Programming (LP) in which term unification is replaced by constraint solving over a specific domain. This brings a number of theoretical and practical advantages which include increased expressive power and declarativeness, as well as higher performance for certain application domains. The resulting CLP languages allow applying efficient, incremental constraint solving techniques to a variety of problems in a very natural way: constraint solving blends in elegantly with the search facilities and the ability to represent partially determined data that are inherent to logic programming. As a result, many modern Prolog systems offer different constraint solving capabilities.

One of the most successful instances of CLP is the class of constraint logic languages using *Finite Domains* ($\mathcal{FD}$). Finite domains refer to those constraint systems in which constraint variables can take values out of a finite set, typically of integers (i.e., a *range*). They are very useful in a wide variety of problems, and thus many Prolog systems offering constraint solving capabilities include a finite domain solver. In such systems, domain (range) definition constraints as well as integer arithmetic and comparison constraints are provided in order to specify problems.

Since the seminal paper of Van Hentenryck et al. [2], many FD solvers adopt the so-called "glass-box" approach. Our FD Kernel also follows this approach, based on a unique primitive called an *indexical*. High-level constraints are then built/defined in terms of primitive constraints. An indexical has the form `X in r`, where r is a range expression (defined in 2). Intuitively, `X in r` constrains the $\mathcal{FD}$ term ($\mathcal{FD}$ variable or integer) `X` to belong to the range denoted by the term `r`. In the definition of the range special expressions are allowed. In particular, the expressions `max(Y)` and `max(Y)` evaluate to the minimum and the maximum of the range of the $\mathcal{FD}$ variable `Y`, and the expression `dom(Y)` evaluates to the current domain of `Y`. Constrains are solved partially in an incremental using consistency techniques [3] which maintain the constraint network in some coherent state (depending on the arc-consistency algorithm used). This is done by monotone domain shrinking and propagation. When all constraints are placed and all values have been propagated a call is typically made to a *labeling* predicate which performs an enumeration-based search for sets of compatible instantiations for each of the variables that remain not bound to a single value. We refer to [2] for more details regarding indexicals and finite domain constraint solving.

In this paper, we present a new free library for Constraint Logic Programming over Finite Domains, included with the Ciao Prolog system [4]. The library is entirely written in Prolog, leveraging on Ciao's module system and code transformation capabilities in order to achieve a highly modular design without compromising performance. We describe the interface, implementation, and design rationale of each modular component. The library meets several design goals: a high level of modularity, allowing the individual components to be replaced by different versions; high-efficiency, being competitive with other $\mathcal{FD}$ implementations; a glass-box approach, so the user can specify new constraints at different levels; and a Prolog implementation, in order to ease the integration with Ciao's code analysis components. The core is built upon two small libraries which implement integer ranges and closures. On top of that, a *finite domain variable* datatype is defined, taking care of constraint reexecution depending on range changes. These three libraries form what we call the $\mathcal{FD}$ *kernel* of the library. This $\mathcal{FD}$ kernel is used in turn to implement several higher-level finite domain constraints, specified using indexicals. Together with a labeling module this layer forms what we name the $\mathcal{FD}$ *solver*. A final level integrates the CLP($\mathcal{FD}$) paradigm with our $\mathcal{FD}$ solver. This is achieved using attributed variables and a compiler from the CLP($\mathcal{FD}$) language to the set of constraints provided by the solver. It should be noted that the user of the library is encouraged to work in any of those levels as seen convenient: from writing a new range module to enriching the set of $\mathcal{FD}$ constraints by writing new indexicals.

One of the first CLP($\mathcal{FD}$) implementations is the CHIP system [5]. This commercial system follows a typical black-box approach: it consists of a complete solver written in

C and interfaces in an opaque manner to a Prolog engine. This makes it difficult for the programmer to understand what is happening in the core of the system. Also, no facilities are provided for tweaking the solver algorithms for a specific application.

More recent CLP($\mathcal{FD}$) systems such as those in SICStus [6], GNU Prolog [7,8], and B-Prolog [9] are built instead following more the glass-box approach. The basic constraints are decomposed into smaller but highly optimized primitives (typically indexicals). Consequently, the programmer has more latitude to extend the constraints as needed. However, even if such systems can be easily modified/extended at the interface level (e.g., both SICStus and B-Prolog provide way to define new global constraints) they are much harder to modify at the implementation level (e.g., it is not possible to replace the implementation of range).

The Ciao CLP($\mathcal{FD}$) library that we present has more similarities with the one recently developed for SWI Prolog [10]. Both are fully written in Prolog and support unbound ranges. The SWI library is clearly more complete than Ciao's (e.g., it provides some global constraints and always terminating propagation), but it is designed in a monolithic way: it is implemented in a single file, mixing different language extensions (using classical Prolog `term_expansion` mechanisms) while the Ciao library is split in more around 20 modules with a clear separation of the different language extensions [11].

Summarizing, our library differs in a number of ways from other existing approaches:

- First, along with more recent libraries it differs from early systems in that it is written entirely in Prolog. This dispenses with the need for a foreign interface and opens up more opportunities for automatic program transformation and analysis. The use of the meta-predicates `setarg/3` and `call/1` means that the use of Prolog has a minimal impact on performance.
- Second, the library is designed as a set of separate modules. This allows replacing a performance-critical part — like the range code — with a new implementation better suited for it.
- Third, the library supports the "glass-box" approach fully, encouraging the user to access directly the low-level layers for performance-critical code without losing the convenience of the high-level CLP paradigm. Again, the fact that the implementation is fully in Prolog is the main enabler of this feature.
- Lastly, we have prioritized extensibility, ease of modification, and flexibility, rather than micro-optimizations and pure raw speed. However, we argue that our design will accommodate several key optimizations like the ones of [12] without needing to extend the underlying WAM.

The rest of the paper proceeds as follows. In Sec. 2 we present the architecture of the library and the interface of the modules. In Sec. 3 we discuss with an example how to use the glass box approach at different levels for better efficiency in a particular problem, with preliminary benchmarks illustrating the gains. Finally, in Sec. 4 we conclude and discuss related and future work.

2 Architecture of the Ciao CLP($\mathcal{FD}$) Library

The Ciao CLP($\mathcal{FD}$) library consists of seven modules grouped into three logical layers plus two specialized Prolog to Prolog translators. In the definition of these modules and interfaces we profit from Ciao's module system [13] and Ciao's support for assertions [14,4], so that every predicate is correctly annotated with its types and other relevant interface-related characteristics, as well as documentation. The translators are built using the Ciao *packages* mechanism [13], which provides integrated and modular support for syntax modification and code transformations. A description of the user interface for the library along with up-to-date documentation may be found in the relevant part of the Ciao manual.

2.1 The Global Architecture

The global architecture is illustrated in Fig. 1. The kernel layer provides facilities for range handling and propagation chains, which are used for defining finite domain variables — which, as mentioned before, are different from the standard logical variables. The $\mathcal{FD}$ layer defines a finite set of constraints such as `a+b=c/3`, using indexicals. These constraints are translated form their indexical form to a set of instructions of the kernel layer. Labeling and branch-and-bound optimization search modules complete the finite domain solver.

The CLP($\mathcal{FD}$) constraints are translated to $\mathcal{FD}$ constraints by a CLP($\mathcal{FD}$) compiler. We use attributed variables to attach a finite domain variable to every logical variable involved in CLP($\mathcal{FD}$) constraints. Thus, the CLP($\mathcal{FD}$) layer is thin and of very low overhead.

2.2 The Finite Domain Kernel

The finite domain kernel is the most important part of the library. Its implementation freely follows the design of the GNU Prolog $\mathcal{FD}$ solver ([8] provides a general overview of this solver). A finite domain variable is composed of a range and several propagation chains. When the submission of a constraint modifies the range of a finite domain variable, other finite domain variables depending on that range are updated by firing up constraints stored in propagation chains. The propagation events are executed in a synchronous way, meaning that a range change will fail if any of its dependent constraints cannot be satisfied.

The kernel implements arithmetic over ranges (pointwise operations, union, intersection complementation, ...) and management of propagation chains, amounting to the delay of Prolog goals on arbitrary events. These two elements are used to implement the two basic operations of a finite domain variable: `tell` and `prune`. The first one attempts to constrain a variable into a particular range, while the second one (`prune/2`) removes a value form the range of a variable. The variable code inspects the new and old ranges and wakes up the suspended goals on a given variable.

All the data structures are coded in an object-oriented style. Efficient access and inplace update are implemented by using the `setarg/3` primitive. We took special care to use `setarg/3` in a *safe* way to avoid undesired side effects, such as those described by Tarau [15].

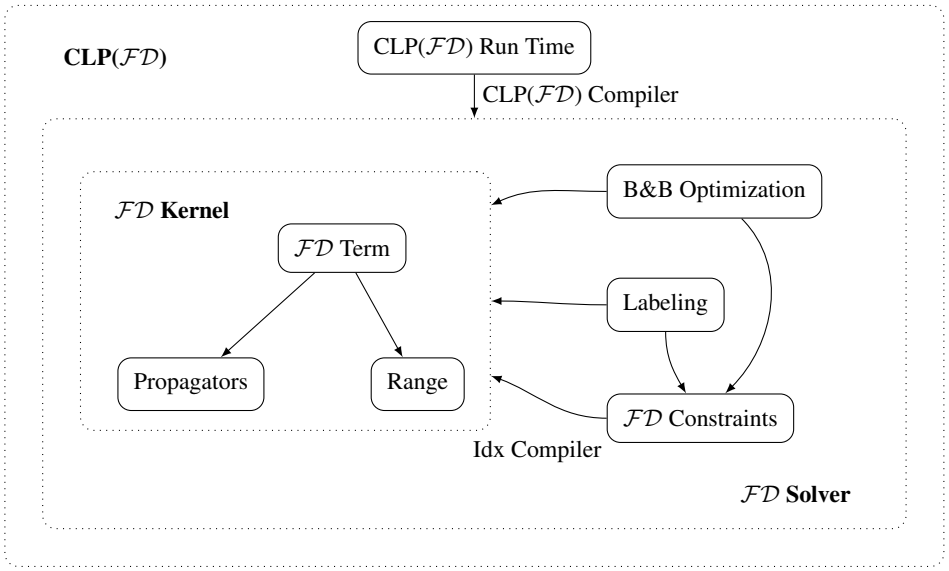


Fig. 1: The Ciao CLP($\mathcal{FD}$) Library Architecture.

2.2.1 Ranges. Range handling is one of the most important parts of the library, given the high frequency of range operations. Indeed, the library supports three implementations for ranges: the standard one using lists of closed integer intervals; an implementation using lists of open (i.e., unbounded) intervals; and a bit-based implementation which despite allowing unbound ranges is more suitable for problems dealing with small ranges.[3] Indeed, the user is encouraged to implement new range modules which are better suited to some particular problems.

The interface that a range module must implement is split into two parts. The first one, shown in Fig. 2, deals with range creation and manipulation. Each of the operations defined in the figure has a corresponding predicate. For instance, bounds addition `t+t` is implemented by the predicate `bound_add/3`, and similarly for the rest of the predicates. Note that it is a convention of the interface that any operation that tries to create an empty range will fail. This is better for efficiency and we found no practical example yet where this would be inconvenient.

Fig. 3 lists the rest of the predicates that a range implementation must provide. They are mainly used for obtaining information about a range and are instrumental for the labeling algorithms.

2.2.2 Propagation Chains. Propagation chains are just lists of goals meant to be executed when a change in the range of a $\mathcal{FD}$ variable happens. The module defines a

[3] The implementation of the bit-based range uses arbitrary precision integers plus three non-ISO predicates for computing the least and most significative bits, and the number of active bits in such integers. We implemented these predicates in C.

`r ::=`	`t .. t`	(interval)
	`{t}`	(singleton)
	`r \/ r`	(union)
	`r /\ r`	(intersection)
	`- r`	(complementation)
	`r + n`	(pointwise addition)
	`r - n`	(pointwise subtraction)
	`r * n`	(pointwise multiplication)
`t ::=`	`min(Y)`	(minimum)
	`max(Y)`	(maximum)
	`dom(Y)`	(domain)
	`val(Y)`	(value)
	`t+t | t-t | t*t | ...`	(arithmetic expression)
	`n`	(bound)

Fig. 2: Range Interface, Part 1: Syntax.

propagation chain structure that is simply a named set of chains. We support in-place update for the structure, thus allowing efficient update of the propagation chains used in the finite domain variables. The interface of the propagation chain module is presented in Fig. 4. We use internal facilities of the Ciao module system in order to efficiently implement `execute/2`.

2.2.3 Finite Domain Variables. An $\mathcal{FD}$ variable is a structure consisting of a range and a propagation chain.

In the current implementation, integers are considered to be finite domain variables too. However, we are in the process of phasing out this optimization as we incorporate more information into finite domain variables to aid optimizations.

$\mathcal{FD}$ variables are never unified, i.e., they cannot be substituted by others or by integer values as is typically done by the Prolog unification mechanism. A priori, such variables have no correspondence to Prolog logical variables.

Apart from accessing its range and propagation chain, the most important operations that a finite domain variable supports is the tell operation, which tries to update the $\mathcal{FD}$ variable to a new range:

`fd_range_bound_t/1`	Type of a range bound.
`fd_range_t/1`	Type of a range object.
`is_singleton/1`	True if range is a singleton.
`singleton_to_bound/2`	Returns the value of a singleton range.
`size/2`	Number of elements in a range.
`get_domain/2`	List of elements in a range.
`enum/2`	Backtracks throughout all the elements in a range.
`bound_const/2`	Correspondence of indexical constants with bounds.

Fig. 3: Range Interface, Part 2: Predicates.

`fd_pchains_t/1`	Type of a chain structure.
`fd_pchain_type_t/1`	Name of a chain.
`empty/1`	Returns an empty chain structure.
`add/3`	Adds a goal to a given chain.
`execute/2`	Wakes up a particular chain.

Fig. 4: Propagation Chain Interface.

```
1 tell_range(FdVar, TellRange):-
2       fd_var:get_range(FdVar, VarRange),
3       fd_range:intersect(VarRange, TellRange, NewRange),
4       set_range_and_propagate(FdVar, VarRange, NewRange)
```

The propagation predicate will set the new range for the variable and compare the new range with the old one. The current definition — following [12] — supports four propagation events, depending on the range change:

dom: The range changed.
max: The maximum of the range has changed.
min: The minimum of the range has changed.
val: The new range is a singleton.

2.3 The Finite Domain Solver

Once the finite domain kernel is in place, the finite domain solver is just the labeling algorithm and a set of constraints defined using the kernel. As mentioned before, the constraints are defined using indexicals, of the form `X in Range`. Such indexicals are compiled to programs of the $\mathcal{FD}$ kernel in a transparent way for the user. The compilation is carried out by Ciao's source-to-source transformation capabilities, which means that an input Prolog file using the `indexicals` package is processed in such a way that predicates containing indexical definitions are replaced by their compiled form.

The indexical syntax is intended to be compatible with syntax used in SICStus and GNU Prolog. However, the use of Ciao's package system means that the user may freely mix indexicals with Prolog code (or with many other syntax extensions, such as, e.g., functional notation) without any ill effect, as seen in Appendix A.

2.3.1 The Constraints Library. A reasonable set of local constraints is provided, covering most examples that we have tried to date. We use the convention of using `t` for ground terms, such that in the constraint `'a+b<>c'/3`, all three arguments are assumed to be $\mathcal{FD}$ variables, whereas in the constraint named `a+t<>c/3`, the second argument is assumed to be an immutable singleton, and thus no propagation chains will be installed on it.

2.3.2 Labeling and Optimization Searches, This layer includes also typical labeling algorithms and branch and bound optimization searches. In fact, the current labeling engine is a slight adaptation of the one in the SWI CLP($\mathcal{FD}$) library: we opted for replacing the preliminary version of the engine with this one from SWI, because of its many useful features and easy adaptability to our library.[4] The porting task was relatively easy because the labeling engine is a quite peripheral part of the library (i.e., it has very few code dependencies). It also underlines the high modularity of our library, since two versions of the labeling are in fact available [5]. Finally we obtained for free a common user interface with SWI (and Yap).

The optimization searches uses a branch-and-bound algorithm with restart to find a value that minimizes (or maximizes) the $\mathcal{FD}$ variable according the execution of a Prolog goal. It offers a user-interface similar to the one provided by GNU Prolog.

2.4 CLP($\mathcal{FD}$)

With the $\mathcal{FD}$ solver in place, supporting the CLP($\mathcal{FD}$) paradigm is a matter of performing two mappings: logical variables must be put in correspondence with $\mathcal{FD}$ variables and CLP($\mathcal{FD}$) constraints must be translated to $\mathcal{FD}$ constraints.

2.4.1 Variable Wrapping. For every logical variable to be involved in a CLP($\mathcal{FD}$) constraint we will attach to it an attribute containing an $\mathcal{FD}$ variable:

```
1  wrapper(A, X):- get_attr_local(A, X), !.
2  wrapper(A, X):- var(A), !, fd_term:new(X), put_attr_local(A, X).
3  wrapper(X, X):- integer(X), !.
```

Logical variables and finite domain variables may communicate in two ways. In the first one, two logical variables may be unified, needing to link their underlying finite domain variables. We implement this communication using the `unify_hook` attribute:

```
1  attr_unify_hook(IdxVar, Other):-
2        ( nonvar(Other) ->
3           ( integer(Other) ->
4              fd_constraints:'a=t'(IdxVar, Other)
5           ; clpfd_error(type_error(Other), '='/2)
6           )
7        ; get_attr_local(Other, IdxVar_) ->
8           fd_constraints:'a=b'(IdxVar, IdxVar_)
9        ; put_attr_local(Other, IdxVar)
10       ).
```

[4] Some features of this engine are currently disabled, but we are planning to activate all such features shortly. The labeling engine was in fact extracted from the *tor* library [16], where it is isolated in a single file.

[5] The old labeling engine can be found in revisions older than 14721 of Ciao 1.15.

We simply call the $\mathcal{FD}$ constraints `'a=b'/2` and `'a=t'/2`.

The other form of communication is instantiation of the logical variable when the corresponding finite domain one gets a singleton range. We modify the `wrapper` predicate to add an instantiation goal to the val chain of freshly created $\mathcal{FD}$ vars, i.e., we replace the second clause within the definition of the wrapper by the following one:

```
1 wrapper(A, X):- var(A), !, fd_term:new(X), put_attr_local(A, X),
2       % Force instantiation of A when X represents an integer
3       fd_term:add_propag(X, val, 'fd_term:integerize'(X, A)).
```

This small example points out the possibilities of our scheme beyond the current use as a support for indexicals.

2.4.2 Constraint Compilation. The $\mathcal{FD}$ solver provides a finite set of $\mathcal{FD}$ constraints, however, in the CLP($\mathcal{FD}$) side we may encounter constraints such as:

```
1     A #= B + C + D + E
```

which should be linearized to

```
1     A1 #= D + E,
2     B1 #= B + C,
3     A #= A1 + B1
```

and then wrapped to:[6]

```
1     'a=b+c'(~wrapper(A1), ~wrapper(D), ~wrapper(E)),
2     'a=b+c'(~wrapper(B1), ~wrapper(B), ~wrapper(C)),
3     'a=b+c'(~wrapper(A) , ~wrapper(A1), ~wrapper(B1))
```

We provide a small compiler which takes care of this process, along with other features like compile-time integer detection.

3 Glass-Box Programming

As previously stated, we encourage the use of a glass box approach when programming with this library. We will use the classical queens program in order to illustrate some of the possibilities that the library offers:

- The use of different range implementations.
- The direct use of the $\mathcal{FD}$ constraints, skipping the CLP($\mathcal{FD}$) compiler.
- The definition of new $\mathcal{FD}$ constraints using indexicals.
- The definition of new atomic constraints directly using the solver kernel, thus skipping the indexical compiler.

[6] We profit here from Ciao's functional notation such that for `p(X,Y)`, `~p(X)` is handled syntactically like a function with return value `Y`.

Queens Parameters	Bits	Closed	Open	SWI
n=16, step, clpfd	0.916	1.144	1.432	1.050
n=16, step, fd	0.572	0.848	1.104	–
n=16, step, idx	0.388	0.648	0.916	–
n=16, step, kernel	0.224	0.336	0.368	–
n=90, ff, clpfd	2.080	2.052	2.484	1.071
n=90, ff, fd	1.112	1.272	1.592	–
n=90, ff, idx	0.752	1.124	1.588	–
n=90, ff, kernel	0.388	0.408	0.432	–

Fig. 5: Queens Benchmark.

Benchmarking Conditions: We provide for illustration purposes some preliminary experimental results. However, it is important to point out that the library is not yet in a state in which relevant absolute performance numbers can be produced and its performance potential fully assessed, since it is still missing important optimizations. Also, only two benchmarks are used.

The benchmarks were run using Ciao 1.15 (revision 14744) on an Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz computer. For reference, we include also the corresponding numbers for SWI Prolog (v. 5.10.4). The purpose is not to make an extensive comparison[7] but rather to have a simple, well understood baseline with which to compare. We should note that we did not explore SWI's support for custom constraints. At the same time, during these tests we have determined that backtracking over changes made by `setarg/3` is currently significantly slower in Ciao than in SWI, which, given the reliance of the implementation on `setarg/3` gives us a clear avenue for performance improvement, independently of any changes to the library itself.

The complete program used in the benchmark is shown in Appendix A. Basically a benchmark has three run time parameters, the number of queens (n=N), the labeling strategy (either "step" or "first fail"[8]), and the constraints used, whose meaning will be explained later. For SWI, only the first two parameters carry significance.

3.1 Range Implementations

As previously stated, the library provides three range implementations, selectable at compile-time. The standard one is called "Closed," and represents ranges using a Prolog list of intervals of integers. Thus, every $\mathcal{FD}$ variable is always bound. "Open" is a variation of this approach where the intervals are enriched with constants `sup` and `inf`. This imposes a penalty on bound arithmetic. Lastly, we compare both against a simple bit-vector implementation, done mostly in Prolog with a small support from C. The results can be seen in Fig. 5. The differences go from negligible to more than 50%.

[7] This is left as future work where, in addition to implementing the optimizations mentioned, we will include comparison with a number of other systems as well.

[8] Comparing the Ciao and SWI libraries using the heuristic labeling strategies as "first fail" is relevant since both use the same code for labeling.

In a different benchmark (bridge), the closed interval version was 25% faster than the open one.

3.2 Constraint Implementations

We now focus on the different possibilities that the library allows for $\mathcal{FD}$ constraint programming.

In the queens program, the main constraint of the problem is expressed by the `diff/3` constraint:

```
1 diff(X, Y, I) :-
2    X #\= Y,
3    X #\= Y+I,
4    X+I #\= Y.
```

where `I` will be always an integer.

However, the compiler cannot (yet) detect that `I` is an integer, and may perform some unnecessary linearization. We may skip the compiler and define `diff` using directly the $\mathcal{FD}$ constraints:

```
1 diff(X, Y, I):-
2    fd_constraints:'a<>b'(~w(X),~w(Y)),
3    fd_constraints:'a<>b+t'(X, Y, I),
4    fd_constraints:'a<>b+t'(Y, X, I).
```

The speedup is considerable, getting close to 50% speedup in some cases. Indeed, the compiler should be improved to produce this kind of code by default.

The user may notice that the above three constraints may be encoded by using just two indexicals. For instance one can use the following definition for `diff/3`:

```
1 diff(X,Y,I):-
2    idx_diff(~w(X), ~w(Y), I).
3 idx_diff(X, Y, I) +:
4    X in -{val(Y), val(Y)+c(I), val(Y)-c(I)},
5    Y in -{val(X), val(X)+c(I), val(X)-c(I)}.
```

Again, the improvement is up to 40% from the previous version.

However, the constraint `diff` can be improved significantly by using directly the kernel delay mechanism (val chain) and $\mathcal{FD}$ variable operations. In particular, we use the optimized kernel `prune/2` operation that removes a single element form the range of a variable:

```
1 diff(X, Y, I):-
2       wrapper(X, X0), wrapper(Y, Y0)
3       fd_term:add_propag(Y, val, 'queens:cstr'(X0, Y0, I)),
4       fd_term:add_propag(X, val, 'queens:cstr'(Y0, X0, I)).
5
6 % Y is always a singleton.
```

```
7  cstr(X, Y, I):-
8        fd_term:integerize(Y, Y0),
9        fd_term:prune(X, Y0),
10       Y1 is Y0 + I,
11       fd_term:prune(X, Y1),
12       Y2 is Y0 - I,
13       fd_term:prune(X, Y2).
```

We reach around 80% speedup from the first version, and this result is optimal regarding what the user can do. Additional speedups can be achieved, but not without going beyond our glass-box approach. Indeed, our CLP($\mathcal{FD}$) compiler is simpler given that we are working on a new translator that directly generates custom kernel constraints from CLP($\mathcal{FD}$) constraints.

4 Conclusions and Future Work

The Ciao CLP($\mathcal{FD}$) library described is distributed with the latest Ciao version, available at `http://ciaohome.org`. Although included in the main distribution, it lives in the `contrib` directory, as it should be considered at a beta stage of development.

Even if we did not include yet important optimizations that should improve significantly the performance of the library, the current results are encouraging. The library has been used successfully internally within the Ciao development team in a number of projects.

The modular design and low coupling of components allow their easy replacement and improvement. Indeed, every individual piece may be used in a glass-box fashion. We expect that the use of Prolog will allow the integration with Ciao's powerful static analyzers. At the same time, the clear separation of run-time and compile-time phases allows the modification and the improvement of the translation schemes in an independent manner. Indeed, the advantages of this design have already been showcased in [17], where a Prolog to Javascript cross-compiler was used to provide a JS version of the library and which only required replacing a few lines of code. Using this cross-compiler CLP($\mathcal{FD}$) programs can be run on the server side or on the browser side unchanged.

Regarding future work, we distinguish two main lines: the kernel and the CLP($\mathcal{FD}$) compiler.

For the kernel, the first priority is to finish settling down its interface. While we consider it mature, some optimizations — like avoiding reexecution — may require that we include more information in our $\mathcal{FD}$ variable structure, range modification times, etc. Indeed, we would like to support more strategies for propagators than the current linear one. Support for some global constraints is on the roadmap, and will likely mean the addition of more propagation chains.

The library features primitive but very useful statistics. However we think it is not enough and we are working on an $\mathcal{FD}$ instrumentation package that will provide detailed statistics and profiling. This is key in order to extract the maximum performance from the library. Once we get detailed profiling information from a wide variety of benchmarks, a better range implementation will be due.

Regarding the CLP($\mathcal{FD}$) compiler, the current version should be considered a proof of concept. Indeed, we are studying alternative strategies including the generation of custom kernels or specialized $\mathcal{FD}$ constraints for each particular program in contrast to the current approach of mapping a CLP($\mathcal{FD}$) program to a fixed set of primitive constraints. CiaoPP — Ciao's powerful abstract interpretation engine — could be used in the translation, providing information about the CLP($\mathcal{FD}$) program to the CLP($\mathcal{FD}$) compiler so it can generate an optimal kernel of $\mathcal{FD}$ code for that program. In this sense, we think that we will follow the CiaoPP approach of combining inference with user-provided annotations in the new CLP($\mathcal{FD}$) compiler.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments.

The research leading to these results has received funding from the Madrid Regional Government under CM project P2009/TIC/1465 (PROMETIDOS), and the Spanish Ministry of Economy and Competitiveness under project TIN-2008-05624 *DOVES*. The research by Rémy Haemmerlé has also been supported by PICD, the Programme for Attracting Talent / young PHDs of the Montegancedo Campus of International Excellence.

References

1. Jaffar, J., Maher, M.: Constraint LP: A Survey. JLP **19/20** (1994) 503–581
2. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation and evaluation of the constraint language cc(fd). Journal of Logic Programming **37**(1–3) (1998) 139–164
3. Dib, M., Abdallah, R., Caminada, A.: Arc-consistency in constraint satisfaction problems: A survey. In: Second International Conference on Computational Intelligence, Modelling and Simulation. (2010) 291–296
4. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming **12**(1–2) (January 2012) 219–252 http://arxiv.org/abs/1102.5497.
5. Dincbas, M., Hentenryck, P.V., Simonis, H., Aggoun, A.: The Constraint Logic Programming Language CHIP. In: Proceedings of the 2nd International Conference on Fifth Generation Computer Systems. (1988) 249–264
6. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education. PLILP '97, London, UK, UK, Springer-Verlag (1997) 191–206
7. D. Diaz, S.A., Codognet, P.: On the implementation of GNU Prolog. Theory and Practice of Logic Programming **12**(1–2) (January 2012) 253–282
8. Codognet, P., Diaz, D.: Compiling constraints in clp(fd). J. Log. Program. **27**(3) (1996) 185–226
9. Zhou, N.F.: Programming finite-domain constraint propagators in action rules. Theory Pract. Log. Program. **6**(5) (September 2006) 483–507
10. Triska, M.: The finite domain constraint solver of swi-prolog. In Schrijvers, T., Thiemann, P., eds.: Functional and Logic Programming. Volume 7294 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2012) 307–316

11. Morales, J.F., Hermenegildo, M.V., Haemmerlé, R.: Modular Extensions for Modular (Logic) Languages. In: 21th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11), Odense, Denmark (July 2011) To appear.
12. Díaz, D., Codognet, P.: A Minimal Extension of the WAM for `clp(fd)`. In: Proceedings of the Tenth International Conference on Logic Programming, Budapest, MIT press (June 1993) 774–790
13. Cabeza, D., Hermenegildo, M.: A New Module System for Prolog. In: International Conference on Computational Logic, CL2000. Number 1861 in LNAI, Springer-Verlag (July 2000) 131–148
14. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58**(1–2) (2005) 115–140
15. Tarau, P.: BinProlog 2006 version 11.x Professional Edition User Guide. BinNet Corporation. (2006) Available from `http://www.binnetcorp.com/`.
16. Schrijvers, T., Triska, M., Demoen, B.: Tor: Extensible Search with Hookable Disjunction. Draft. Available from `http://users.ugent.be/~tschrijv/tor/` (2012)
17. Morales, J.F., Haemmerlé, R., Carro, M., Hermenegildo, M.V.: Lightweight compilation of (C)LP to JavaScript. Theory and Practice of Logic Programming, 28th Int'l. Conference on Logic Programming (ICLP'12) Special Issue (2012) To appear.

A Complete Code for the Queens Example

```
1  queens(N, L, Lab, Const) :-
2        length(L, N),
3        domain(L, 1, N),
4        safe(L, Const),
5        labeling(Lab, L).
6
7  safe([], _Const).
8  safe([X|L], Const) :-
9        noattack(L, X, 1, Const),
10       safe(L, Const).
11
12 noattack([], _, _, _Const).
13 noattack([Y|L], X, I, Const) :-
14       diff(Const, X, Y, I),
15       I1 is I + 1,
16       noattack(L, X, I1, Const).
17
18 diff(clpfd, X, Y, I) :-
19       X #\= Y, X #\= Y+I, X+I #\= Y.
20
21 diff(fd, X,Y,I):-
22       fd_diff(~wrapper(X), ~wrapper(Y), I).
23
24 fd_diff(X, Y, I):-
25       fd_constraints:'a<>b'(X,Y),
26       fd_constraints:'a<>b+t'(X,Y,I),
```

```
27      fd_constraints:'a<>b+t'(Y,X,I).
28
29  diff(idx, X,Y,I):-
30      idx_diff(~wrapper(X), ~wrapper(Y), I).
31
32  idx_diff(X, Y, I) +:
33      X in -{val(Y), val(Y)+c(I), val(Y)-c(I)},
34      Y in -{val(X), val(X)-c(I), val(X)+c(I)}.
35
36  diff(kernel, X,Y,I):-
37      kernel_diff(~wrapper(X), ~wrapper(Y), I).
38
39  kernel_diff(X, Y, I) :-
40      fd_term:add_propag(Y, val, 'queens:cstr'(X, Y, I)),
41      fd_term:add_propag(X, val, 'queens:cstr'(Y, X, I)).
42
43  cstr(X, Y, I):-
44      fd_term:integerize(Y, Y0),
45      fd_term:prune(X, Y0),
46      Y1 is Y0 + I, fd_term:prune(X, Y1),
47      Y2 is Y0 - I, fd_term:prune(X, Y2).
```