YapTab: A Tabling Engine Designed to Support Parallelism

Ricardo Rocha Fernando Silva Vítor Santos Costa

Abstract

This paper addresses the design and implementation of YapTab, a tabling engine that extends the Yap Prolog system to support sequential tabling. The tabling implementation is largely based on the XSB engine, the SLG-WAM, however substantial differences exist since our final goal is to support parallel tabling execution. We discuss the major contributions in YapTab and outline the main differences of our design in terms of data structures and algorithms. Finally, we present some initial performance results for YapTab and compare with those for XSB.

Keywords: Logic Programming, Tabling, Implementation, Performance.

1 Introduction

Logic programming systems provide a high-level, declarative approach to programming. Arguably, Prolog is the most popular logic programming language. In Prolog programs are written using Horn Clauses, a subset of First-Order Logic that has an intuitive interpretation as positive facts and rules. The operational semantics of Prolog is given by SLD-resolution, a refutation strategy particularly simple and that matches stack-based machines particularly well.

Ideally one would want Prolog programs to be written as logical statements, and for control to be a separate issue. In practice, Prolog programmers must be aware of the Prolog computation rule, as the limitations of the Prolog computation rule mean that logically correct programs can, say, enter infinite loops. To improve the declarativeness and expressiveness of Prolog several proposals have been put forth. One such proposal that has been gaining in popularity is the use of *tabling.* In a nutshell, tabling is about storing intermediate solutions to a query so that they can

Ricardo Rocha and Fernando Silva are with DCC-FC & LIACC, University of Porto, Portugal. E-mails: {ricroc,fds}@ncc.up.pt

Vítor Santos Costa is with COPPE Systems, Federal University of Rio de Janeiro, Brazil. E-mail: vitor@cos.ufrj.br

Work partially supported by the projects Melodia (JNICT PBIC/C/TIT/2495/95), Dolphin (PRAXIS/2/2.1/TIT/1577/95), CLoP (CNPq), PLAG (FAPERJ) and by Fundação para a Ciência e Tecnologia.

be reused later. Work on SLG-resolution [1] as implemented in the XSB System [14] proved the viability of this technology.

One important advantage of tabling systems is that they avoid looping and thus terminate for all programs with the *bounded term-size property* [2]. This removes one of the main limitations of Prolog as a query processing language for data-base processing. Tabling can be seen as a natural way for storing intermediate results, nicely matching the characteristics of well-known algorithms for natural language processing and program analysis, just to mention a few examples. The fundamental robustness of tabling and of SLG-resolution has also made it a good basis for extending Prolog's Horn clause language, including support for negation.

Although tabling can work for both deterministic and non-deterministic programs, quite a few interesting applications of tabling are by nature non-deterministic. This rises the question of whether further efficiency would be possible through the use of or-parallelism, that is, by running several branches of the search tree in parallel. In previous work [11] we proposed two computational models, the *Or-Parallelism within Tabling (OPT)* and *Tabling within Or-Parallelism (TOP)* models, to combine tabling with or-parallelism. We have since decided to implement the OPT model [12] over the YapOr system [13], based on the high-performance Yap Prolog compiler [4].

This paper addresses the implementation of YapTab, a sequential tabling engine designed to support or-parallelism. The implementation is based on the ground-breaking work for the XSB system. We innovated by considering the novel issues arising with the introduction of parallelism. In terms of the basic engine, the original XSB design was therefore changed when restoring computations, determining leader nodes and completing subgoals.

The remainder of the paper is organized as follows. First, we briefly introduce the tabling concepts and the SLG-WAM. Next, we present the OPT computational model and discuss the major contributions in YapTab. We then present the main data areas, data structures and algorithms to extend the Yap Prolog system to support tabling. Last, we present some early performance data and terminate by outlining some conclusions and further work.

2 Tabling Concepts and the SLG-WAM

Tabling is about storing and reusing intermediate answers for goals. Whenever a tabled subgoal S is called for the first time, an entry for S is allocated in the *table space.* This entry will collect all the answers generated for S. Repeated calls to variants of S are resolved by consuming the answers already stored in the table. Meanwhile, as new answers are generated for S, they are inserted into the table and returned to all variant subgoals. Within this model, the nodes in the search space are classified as either *generator nodes*, corresponding to first calls to tabled subgoals, *consumer nodes*, that consume answers from the table space, and *interior nodes*, that are evaluated by standard SLD-resolution.

Space for a subgoal can be reclaimed when the subgoal has been *completely evaluated.* A subgoal is said to be completely evaluated when all its possible resolutions have been performed, that is, when no more answers can be generated and the variant subgoals have consumed all the available answers. Note that a number of subgoals may be mutually dependent, forming a *strongly connected component* (or *SCC*) [14], and therefore can only be completed together. The completion operation is thus performed by the *leader* of the SCC, that is, by the oldest subgoal in the SCC, when all possible resolutions have been made for all subgoals in the SCC. Hence, in order to efficiently evaluate programs one needs an efficient and dynamic detection scheme to determine when all the subgoals in a SCC have been completely evaluated.

For definite programs, tabling based evaluation has four main types of operations: *Tabled Subgoal Call* creates a generator node; *New Answer* verifies whether a newly generated answer is already in the table, and if not, inserts it; *Answer Resolution* consumes an answer from the table; and *Completion* determines whether an SCC is completely evaluated, and if not, schedules a resolution to continue the execution.

In XSB, the implementation of tabling was attained by extending the WAM [15] into the SLG-WAM, with minimal overhead. In short, the SLG-WAM introduces special instructions to deal with the operations above and two new memory areas: a *table space*, used to save the answers for tabled subgoals; and a *completion stack*, used to detect when a set of subgoals is completely evaluated.

Further, whenever a consumer node gets to a point in which it has consumed all available answers, but the correspondent tabled subgoal has not yet completed and new answers may still be generated, the computation must be suspended. In the SLG-WAM the suspension mechanism is implemented through a new set of registers, the *freeze registers*, which freeze the WAM stacks at the suspension point and prevent all data belonging to the suspended branch from being erased. To later resume a suspended branch, the bindings belonging to the branch must be restored. SLG-WAM achieves this by using an extension of the standard trail, the *forward trail*, to keep track of the bindings values. Other implementation mechanisms for tabling have been proposed recently [6, 5, 9, 16].

3 Tabling and Parallelism

In previous work [11] we proposed two computational models to combine tabling with or-parallelism, the OPT and the TOP approaches. We have decided to implement the OPT approach [12]. The OPT approach generalizes Warren's multi-sequential engine framework for or-parallelism. The or-parallelism stems from having several engines that implement SLG-resolution, instead of implementing Prolog's SLD-resolution.

Tabling is the base component of the OPT computational model. Each computational agent, or *worker*, can be considered a full sequential tabling engine and should spend most of its computation time exploiting the search tree involved in such an evaluation. It allocates all three types of nodes, fully implements suspension of tabled subgoals, and resumes subcomputations to consume newly found answers. Or-parallelism is only triggered when a worker runs out of alternatives to exploit. Unexploited alternatives should be made available for parallel execution, regardless of whether they originate from a generator, consumer or interior node. Therefore, parallelism stems from both tabled and non-tabled subgoals. This contrasts with the table-parallelism model [7] that only considers tabled subgoals as candidates for parallel execution.

To implement the OPT approach, we have decided to use the YapOr system [13] as the parallel component of the model. The YapOr system is an or-parallel Prolog system based on environment copying. Thus, we have designed the YapTab tabling engine in order to meet the requirements of the OPT approach based on environment copying. Although the YapTab tabling engine is SLG-WAM based, as is the XSB system, we have taken some different implementation decisions in order to avoid potential sources for execution overhead resulting from its integration with the environment copying model.

In an environment copying model, sharing is implemented through copying of the execution stacks between workers and, thus, a shared branch may exist several times for the workers that are sharing the branch. The duplication of items is a major source of overhead. It implies larger stack areas to be copied when sharing, and it requests synchronization mechanisms when updating common items and when replicating the new values. Hence, in order to efficiently integrate the tabling and the or-parallel components of the OPT model, we should minimize this duplication. To address this need, YapTab introduces a new data structure, the *dependency frame*, that resides in a single shared space that we called the *dependency space*.

The dependency frame data structure keeps track of all data related with tabling suspensions. This allows us to reduce the number of extra fields in tabled choice points and to eliminate the need for a completion stack area. Moreover, a smaller number of extra fields in tabled choice points minimizes the time needed to perform copying. Eliminating the completion stack area from the YapTab design reduces the number of stack areas to be copied when sharing, and simplifies the complexity in managing shared tabling suspensions.

In practice, we found that this solution simplifies the parallel implementation of fundamental aspects to the system's efficiency. Sharing tabling suspensions is straightforward, the worker requesting work only needs to update its private top dependency frame pointer to the one's of the sharing worker. Concurrent accesses or updates to the shared suspension data can be synchronized through the use of a locking mechanism at the dependency frame level. The OPT completion algorithm [12] for shared branches is mainly based on the dependency frame data structure which avoids explicit communication and synchronization between workers.

4 Extending Yap to Support Tabling

The YapTab design is WAM based, as is the SLG-WAM. It implements two tabling scheduling strategies, batched and local [8], and in the initial design it only considers table predicates without any kind of negative calls.

As in the original SLG-WAM, we introduce a new data area, the table space; a new set of registers, the freeze registers; an extension of the standard trail, the forward trail; and the four main tabling operations: tabled subgoal call, new answer, answer resolution and completion. As the basis for tables we use tries as proposed in [10]. Tries provide complete discrimination for terms and permit a lookup and possible insertions to be performed in a single pass through a term, which makes it easily parallelizable.

The substantial differences between the two designs, and corresponding implementations, reside in the aspects that can be a potential source of overheads when the tabling engine is extended to a parallel model. In order to efficiently integrate the tabling and the or-parallel components of the OPT computational model, YapTab introduces the dependency frame data structure. To take advantage of the philosophy behind the dependency frame data structure, all the algorithms related with suspension, resumption and completion were redesigned. We then present the main data areas, data structures and algorithms implemented to extend the Yap system to support tabling.

Tabled Nodes

Remember that interior nodes correspond to normal (not tabled) subgoals and they are evaluated by standard SLD-resolution. Generator and consumer nodes correspond, respectively, to first and variant calls to tabled subgoals. The generator nodes use program clause resolution to produce and store answers in the table space for the corresponding tabled subgoal. The consumer nodes load from the table space the answers previously stored by the associated generator node.

Interior nodes are implemented as normal WAM choice points (we assume familiarity with WAM choice points [15]). In the SLG-WAM, generator nodes are WAM choice points extended with a few extra fields to control the tabling execution. In our implementation the generator choice point only requires one of the extra fields used in the SLG-WAM, that is a pointer to the associated *subgoal frame* [14] in the table space, hence allowing correct access to the chain of answers associated with the subgoal. Figure 1 illustrates the type of nodes being considered and their relationship with the table and dependency spaces.

In the SLG-WAM, the consumer choice points store supplementary information about the suspension point. In our case, we move that information to a dependency frame and leave a pointer to this frame in the consumer choice point. Hence, the consumer choice point only requires an extra field. Each dependency frame is linked to the previous one forming a dependency chain of consumer nodes. Additionally, the dependency frame stores information to efficiently check for completion points, and to efficiently move across the consumer nodes with unconsumed answers. As we shall see, this additional information replaces the need for a completion stack.

Freeze Registers

A tabling evaluation can be seen as a sequence of suspension and resumptions of subcomputations. To preserve the environment of a suspended computation the stacks are frozen using a set of freeze registers, one per stack. This way, and until the completion of the appropriate subgoal call takes place, the space belonging to the suspended branch is safe from being erased. It is only upon completion that we can release the space previously frozen and adjust the freeze registers. In our

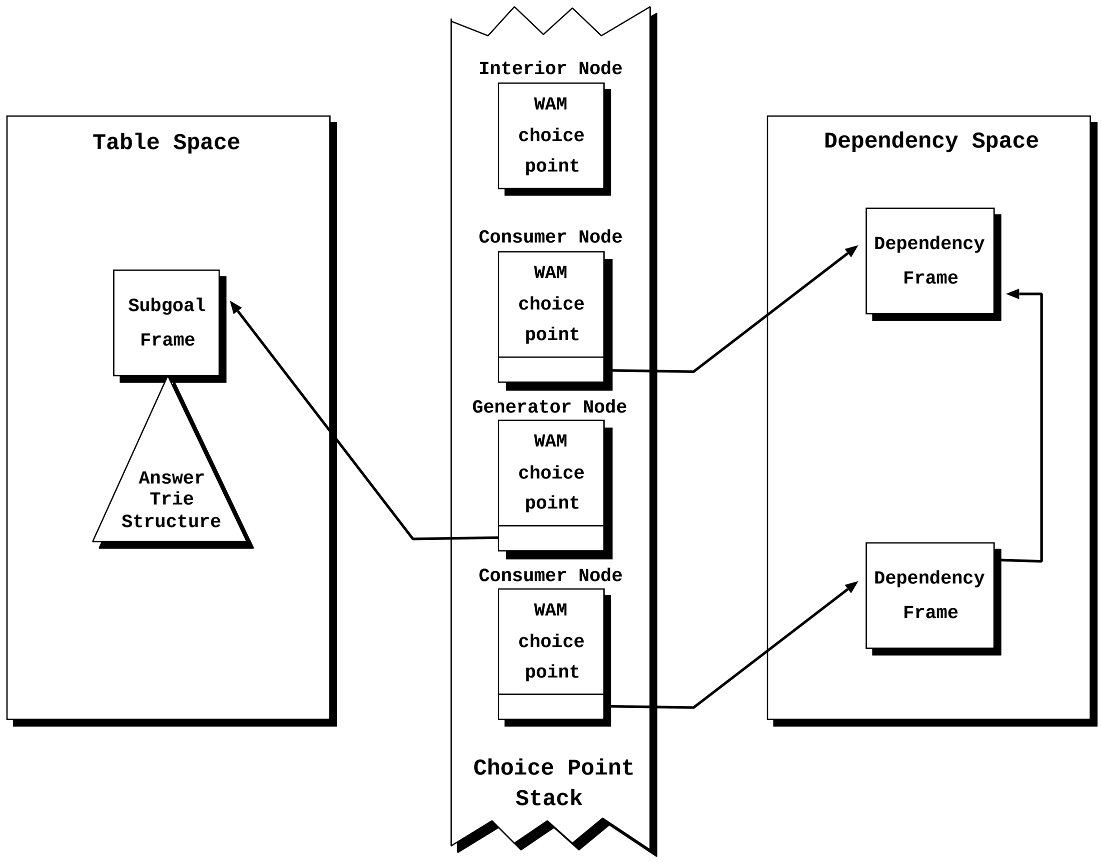

Figure 1: The nodes and their relationship with the table and dependency spaces.

approach, we adjust the freeze registers by using the top stack pointers saved in the last suspension point, that is, the younger consumer node in the currently active stacks. To access that node we check for the top dependency frame in the chain of dependency frames. It is interesting to remark that we apply exactly the same mechanism for our implementation of or-parallelism with the SBA model [3].

Completion and Leader Nodes

The completion operation takes place when a generator node exhausts all alternatives and it finds itself as a leader node. We designed our algorithms to quickly determine whether a generator node is a leader node. A special field in the dependency frame structure is used to hold a pointer to the leader node of the SCC that includes the current suspension point. To compute the leader node information when allocating a dependency frame, we first hypothesize that the leader node is the generator node for the current variant subgoal call, say $\mathcal{N}$. Next, for all consumer nodes between $\mathcal{N}$ and the current consumer node, we check whether they depend on an older generator node. Consider that the oldest dependency is for the generator node $\mathcal{N}'$. If this is the case, then $\mathcal{N}'$ is the leader node, otherwise our hypothesis was correct and the leader is indeed the initially found generator node $\mathcal{N}$.

By using the leader node information from the dependency frames, the generator nodes can quickly determine whether they are leader nodes. A generator node finds itself as a leader node if there are no younger dependencies (i.e., no younger consumer nodes) or if it is the leader node referred in the top dependency frame. Figure 2 illustrates a small example of node dependencies.

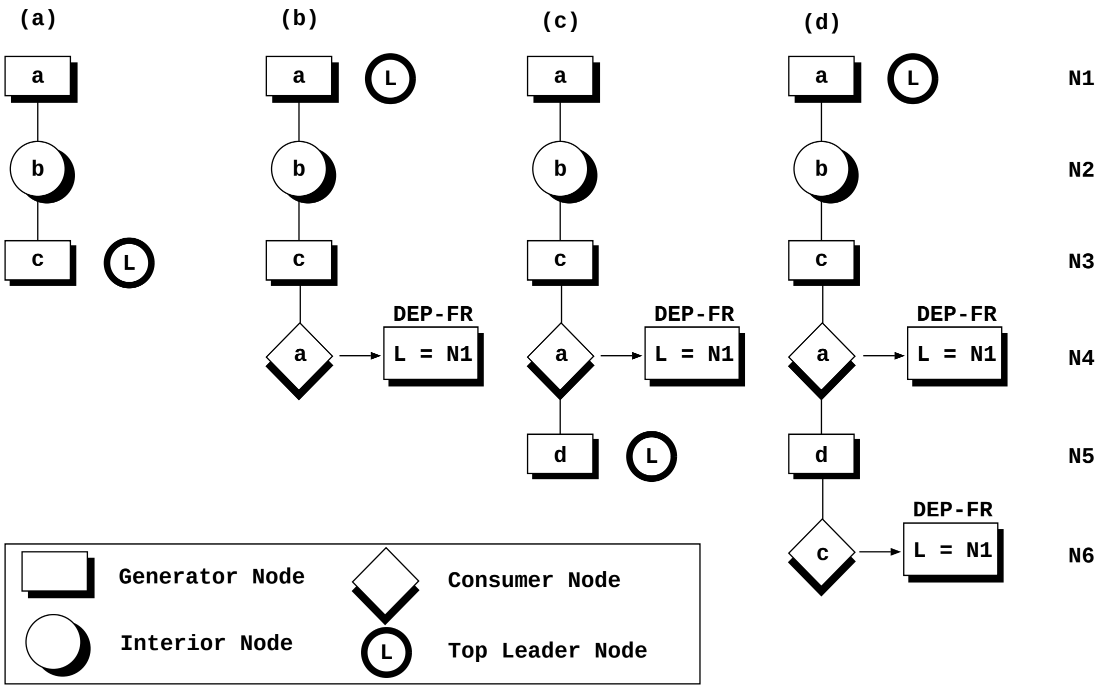

Figure 2: Spotting the top leader node.

In Fig. 2(a), the generator node N3 is the top leader node because there are no younger consumer nodes. By top leader node we mean the generator node where the next completion operation may take place, i.e., the younger generator node that is a leader node. Suppose that a new consumer node is created, that is node N4 in Fig. 2(b). Then a dependency frame associated with N4 has to be allocated and the leader node information must be computed. In this case, the leader node is N1 because N4 is a variant subgoal a of the generator node N1 and there are no other consumer nodes in between. As a result, the top leader node for the set of nodes including N4 becomes N1. Figure 2(c) illustrates a similar case to the first one, and N5 becomes the new top leader node. The consumer node N6 (Fig. 2(d)) for the variant subgoal c has its generator node at node N3. Since in between nodes N6 and N3 there is a dependency for an older generator node, N1, given by the frame associated with consumer node N4, the leader node information for the dependency frame associated with N6 is also N1. This turns N1 again as the top leader node.

Whenever a generator node finds that it is the top leader node, it starts to check if there are younger consumer nodes with unconsumed answers. This can be done by going through the chain of dependency frames looking for a frame with unconsumed answers. If there is such a frame, it resumes the computation to the corresponding consumer node. Otherwise, we can perform completion. This includes marking as completed all the subgoals in the SCC; deallocating all the younger dependency frames; and readjusting the pointer to the chain of dependency frames.

Answer Resolution

The answer resolution operation has to be performed whenever the computation backtracks or is resumed to a consumer node. That operation is responsible for guaranteeing that every answer is consumed once and just once.

First, the answer resolution operation checks for unconsumed answers. If there are new answers in the table for the subgoal at hand, it loads the next available answer and proceeds the execution. Otherwise, it checks in the dependency frame for a backtracking node. If this is the first time that backtracking from that consumer node takes place, then backtracking is performed as usual to the previous node. Otherwise, we know that the computation has been resumed from an older generator node $\mathcal{N}$ during an unsuccessful completion operation and, therefore, backtracking must be done to the next consumer node that has unconsumed answers and is younger than $\mathcal{N}$. If there are no such consumer nodes then backtracking must be done to the $\mathcal{N}$ generator node.

5 Initial Performance Evaluation

We have implemented two scheduling strategies, batched and local scheduling, that constraint differently the tabling execution. The performance study will be made using both strategies. We start by analyzing the overheads of supporting tabling in Yap on a set of non tabled Prolog programs, and by measuring the XSB behavior on the same set of programs. We then use four versions of a tabled program with varying complexity of the search space to compare YapTab with XSB.

YapTab is based on the Yap4.2.1 engine and the results were obtained on a 200 MHz PentiumPro with 128MB of main memory, a 256KB cache, and running the linux-2.2.5 kernel. We used the same compilation flags for Yap and for YapTab. Regarding XSB, we used version 2.2 with the default configuration and the default execution parameters (chat engine and batched scheduling).

To put the performance results in perspective we first use a standard set of non tabled logic programming benchmarks to compare the performance of YapTab with Yap and XSB. The benchmarks include the n-queens problem, the puzzle and cubes problems from Evan Tick's book, an hamiltonian graph problem and a naïve sorting resolution.

Benchmark	**YapTab**	**Yap Prolog**	**XSB Prolog**
9-queens	740	740(1.00)	1819(2.46)
cubes	210	210(1.00)	589(2.80)
ham	460	430(0.93)	1139(2.48)
nsort	390	370(0.95)	1101(2.82)
puzzle	2430	2120(0.87)	5819(2.39)
Average		(0.95)	(2.59)

Table 1: YapTab, Yap and XSB running times on a set of non tabled benchmarks.

Table 1 shows the base running times, in milliseconds, for YapTab, Yap Prolog and XSB Prolog for the set of non tabled benchmarks. In parentheses, it shows the performance of Yap and XSB over the YapTab running times. The results indicate that YapTab introduce, on average, an overhead of about 5% over standard Yap. These overheads are due to the handling of the freeze registers and the forward trail. Regarding XSB, the results show that, on average, YapTab is 2.59 times faster than XSB, a result mainly due to the faster Yap engine.

To assess the performance of YapTab when running tabled programs and compare it with XSB, we wrote four versions of the following tabled program:

```
:- table path/3.

path(X,Y,[X,Y]) :- arc(X,Y).
path(X,Y,P) :- path(X,Z,P1), arc(Z,Y), insert_if_new(Y,P1,P).

insert_if_new(X,[],[X]).
insert_if_new(X,[H|T],[H|NT]) :- X \= H, insert_if_new(X,T,NT).
```

The four versions differ mainly in the number of nodes and graph topology, hence defining search spaces of varying complexity. In all versions, the query goal is to find the transitive closure of the given graph.

Benchmark	**YapTab Batched**	**YapTab Local**	**XSB Prolog**
binary tree (depth 10)	440	480(1.09)	451(1.03)
chain (64 nodes)	120	150(1.25)	399(3.33)
cycle (64 nodes)	380	470(1.24)	1121(2.95)
grid (4x4 nodes)	1270	1390(1.09)	5740(4.52)
Average		(1.17)	(2.96)

Table 2: YapTab and XSB running times on a four version tabled benchmark.

Table 2 shows the running times, in milliseconds, for YapTab, using batched (YapTab Batched) and local (YapTab Local) scheduling strategies, and XSB Prolog for the four benchmarks. The results show that YapTab Batched on average performs better than YapTab Local and XSB. In parentheses it shows the overheads of YapTab Local and XSB over YapTab Batched. The results obtained for batched and local scheduling confirm previous results obtained for XSB using the same scheduling strategies [8]. The results also show that, for these programs, YapTab is about 3 times faster than current XSB. This is due to the faster Yap engine, as seen in table 1, and to the fact that XSB implements functionalities that are still lacking in YapTab and those may cause overheads during execution. The exception is the binary tree benchmark where XSB benefits from the use of hashing in the implementation of tries to optimize table access [10]. YapTab does not yet implement this hashing optimization and, therefore, its performance degrades with searching through the larger number of different nodes that the binary tree program presents.

6 Conclusions

We have presented the design and implementation of YapTab, an extension of the Yap Prolog system that implements sequential tabling. Our system includes all the machinery required to execute programs with tabling in or-parallel. YapTab reuses the principles of XSB-Prolog's SLG-WAM engine, whilst innovating by separating the tabling suspension data in a single space, the dependency space, and by proposing a new completion detection algorithm not based on the intrinsically sequential completion stack.

Our first results are very encouraging. Overheads over standard Yap are low and performance in tabling benchmarks is quite satisfactory even when compared with the arguably more mature and complete XSB system.

We have obtained very initial timings for parallel execution on a shared memory PentiumPro machine. The results show significant speedups for a tabled application increasing up to the four processors and encourage us in our believe that tabling and parallelism may together contribute to increasing the range of applications for Logic Programming.

References

[1] W. Chen and D. S. Warren. Query Evaluation under the Well Founded Semantics. In *Proceedings of PODS'93*, pages 168–179, 1993.

[2] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[3] V. S. Costa, R. Rocha, and F. Silva. Novel Models for Or-Parallel Logic Programs: A Performance Analysis. In *Proceedings of EuroPar'2000*, September 2000. To appear in Springer-Verlag LNCS series.

[4] L. Damas, V. S. Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual.* Centro de Informática da Universidade do Porto, 1989.

[5] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *Proceedings of PLILP/ALP98*. Springer Verlag, September 1998.

[6] B. Demoen and K. Sagonas. CHAT: The Copy-Hybrid Approach to Tabling. *Lecture Notes in Computer Science*, 1551:106–121, 1999.

[7] J. Freire, R. Hu, T. Swift, and D. S. Warren. Exploiting Parallelism in Tabled Evaluations. In *Proceedings of PLILP'95*, pages 115–132, 1995.

[8] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *Proceedings of PLILP'96*, pages 243–258. Springer-Verlag, Sep. 1996.

[9] Hai-Feng Guo and G. Gupta. A New Tabling Scheme with Dynamic Reordering of Alternatives. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages*, July 2000. To appear.

[10] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *Proceedings of ICLP'95*, pages 687–711. The MIT Press, June 1995.

[11] R. Rocha, F. Silva, and V. S. Costa. On Applying Or-Parallelism to Tabled Evaluations. In *Proceedings of the First International Workshop on Tabling in Logic Programming*, pages 33–45, Leuven, Belgium, June 1997.

[12] R. Rocha, F. Silva, and V. S. Costa. Or-Parallelism within Tabling. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages, PADL'99*, number 1551 in LNCS, pages 137–151, San Antonio, Texas, USA, January 1999. Springer-Verlag.

[13] R. Rocha, F. Silva, and V. S. Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence, EPIA'99*, number 1695 in LNAI, pages 178–192, Évora, Portugal, September 1999. Springer-Verlag.

[14] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20, 1998.

[15] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Sep. 1983.

[16] Neng-Fa Zhou. Implementation of a Linear Tabling Mechanism. In *Proceedings of PADL'2000*, number 1753 in LNCS, pages 109–123. Springer-Verlag, January 2000.