# On a Tabling Engine that Can Exploit Or-Parallelism

Ricardo Rocha Fernando Silva Vítor Santos Costa *

**Abstract.** Tabling is an implementation technique that improves the declarativeness and expressiveness of Prolog by reusing solutions to goals. Quite a few interesting applications of tabling have been developed in the last few years, and several are by nature non-deterministic. This raises the question of whether parallel search techniques can be used to improve the performance of tabled applications.
In this work we demonstrate that the mechanisms proposed to parallelize search in the context of SLD resolution naturally generalize to parallel tabled computations, and that resulting systems can achieve good performance on multi-processors. To do so, we present the OPTYap parallel engine. In our system individual SLG engines communicate data through stack copying. Completion is detected through a novel parallel completion algorithm that builds upon the data structures proposed for or-parallelism. Scheduling is simplified by building on previous research on or-parallelism. We show initial performance results for our implementation. Our best result is for an actual application, model checking, where we obtain linear speedups.
**Keywords:** Parallel Logic Programming, Or-Parallelism, Tabling.

## 1 Introduction

The past years have seen wide effort at increasing Prolog's declarativeness and expressiveness. *Tabling* or *memoing* is one such proposal that has been gaining in popularity. In a nutshell, tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears. Work on SLG resolution [3], as implemented in the XSB System [15], proved the viability of tabling technology for application areas such as natural language processing, knowledge based systems, model checking, or program analysis. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property* [4].

Tabling works for both deterministic and non-deterministic applications, but it has frequently been used to reduce search space. This rises the question of whether further efficiency improvements may be achievable through parallelism. Freire and colleagues [7] were the first to propose that tabled goals could indeed be a source of implicit parallelism. In their model, each tabled subgoal is

* R. Rocha and F. Silva are with DCC-FC & LIACC, University of Porto, Portugal. E-mails: {ricroc,fds}@ncc.up.pt. V. S. Costa is with COPPE Systems, University of Rio de Janeiro, Brazil. E-mail: vitor@cos.ufrj.br. Work partially supported by CLoP (CNPq), PLAG (FAPERJ) and by Fundação para a Ciência e Tecnologia.

computed independently in a separate computational thread, a *generator thread.* Each generator thread is the sole responsible for fully exploiting its subgoal and obtain the complete set of answers. This model restricts parallelism to concurrent execution of generator threads. Parallelism arising from non-tabled subgoals or from alternative clauses is not exploited.

Our suggestion is that we should exploit parallelism from both tabled and non-tabled subgoals. By doing so we can both extract more parallelism, and reuse the mature technology for tabling and parallelism. Towards this goal, we previously proposed two computational models to combine tabling with or-parallelism [12], *Or-Parallelism within Tabling (OPT)* and *Tabling within Or-Parallelism (TOP)* models.

This paper presents an implementation for the OPT model, the OPTYap system. To the best of our knowledge, OPTYap is the first available system that can exploit parallelism from tabled programs. The OPT model considers tabling as the base component of the system. Each computational worker behaves as a full sequential tabling engine. The or-parallel component of the system is triggered to allow synchronized access to the shared part of the search space or to schedule work.

From the beginning, we aimed at developing an or-parallel tabling system that, when executed with a single worker, runs as fast or faster than current sequential tabling systems as otherwise, parallel performance would not be significant and fair. To achieve these goals, OPTYap builds on YapOr [13] and YapTab [14] engines. YapOr is an or-parallel engine that extends Yap's efficient sequential engine [16]. It is based on the environment copy model, as first implemented in Muse [1]. YapTab is a sequential tabling engine that extends Yap's execution model to support tabled evaluation. YapTab's implementation is largely based on the ground-breaking SLG-WAM work used in the XSB system [15].

The remainder of the paper is organized as follows. First, we briefly introduce the basic tabling definitions and the SLG-WAM. Next, we present the OPT computational model and discuss its implementation framework. We then present the new data areas, data structures and algorithms to extend the Yap Prolog system to support sequential and parallel tabling. Last, we present some early performance data and terminate by outlining some conclusions and further work.

## 2 Tabling and the SLG-WAM

Tabling is about storing and reusing intermediate answers for goals. In variant-based tabling, 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 found 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, 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*, corresponding to variant calls to tabled subgoals, and *interior nodes*, corresponding to non-tabled predicates.

Tabling based evaluation has four main types of operations for definite programs. The *Tabled Subgoal Call* operation checks if the subgoal is in the table and if not, inserts it and allocates a new generator node. Otherwise, allocates a consumer node and starts consuming the available answers. The *New Answer* operation verifies whether a newly generated answer is already in the table, and if not, inserts it. The *Answer Resolution* operation consumes the next newly found answer, if any. The *Completion* operation determines whether a tabled subgoal is completely evaluated, and if not, schedules a possible resolution to continue the execution.

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*) [15], and therefore can only be completed together. The completion operation is thus performed at 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.

The implementation of tabling in XSB Prolog was attained by extending the WAM [17] into the SLG-WAM [15]. In short, the SLG-WAM introduces a new set of instructions to deal with the operations above, a special mechanism to allow suspension and resumption of computations, 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. The SLG-WAM also introduced the concepts of freeze registers and forward trail to handle suspension [15].

## 3 Or-Parallelism within Tabling

The OPT model [12] divides the search tree into a public and several private regions, one per worker. Workers in their private region execute nearly as in sequential tabling. Workers exploiting the public region of the search tree must be able to synchronize in order to ensure the correctness of the tabling operations. When a worker runs out of alternatives to exploit, it enters in scheduling mode. The YapOr scheduler is used to search for busy workers with unexploited work. Alternatives should be made available for parallel execution, regardless of whether they originate from generator, consumer or interior nodes.

Parallel execution requires significant changes to the SLG-WAM. Synchronization is required **(i)** when backtracking to public generator or interior nodes to take the next available alternative; **(ii)** when backtracking to public consumer nodes to take the next unconsumed answer; or, **(iii)** when inserting new answers into the table space. In a parallel tabling system, the relative positions of generator and consumer nodes are not as clear as for sequential systems. Hence

we need novel algorithms to determine whether a node is a leader node and to determine whether a SCC can be completed.

OPTYap uses environment copying for or-parallelism and the SLG-WAM for tabling because these are, respectively, two of the most successful or-parallel and tabling engines. Copying is a popular and effective approach to or-parallelism that minimizes actual changes to the WAM. To share work we use *incremental copying* [1], that is, we only copy *differences* between stacks.

In contrast to copying, the SLG-WAM requires significant changes to the WAM in order to support freezing of goals. These changes introduce overheads, namely in trailing and in stack manipulation. Demoen and Sagonas addressed the problems by suggesting CAT [5] and more recently, CHAT [6]. These two models reduce overheads by copying parts of stacks, instead of freezing. Although there is an attractive analogy between copying and CAT or CHAT, a more detailed analysis shows significant drawbacks. First, both assume separate choice-point and local stacks. Second, both rely on an incremental saving technique to reduce copying overheads. Unfortunately, the technique assumes that completion always takes place at generator nodes. As we shall see, these assumptions do not hold true for parallel tabling. Last, both may incur in substantial slowdowns for some applications. We therefore used the SLG-WAM in our work.

Rather different approaches to tabling have also been proposed recently [18, 9]. In both cases, the main idea is to recompute tabled goals, instead of suspending. Unfortunately, the process of retrying alternatives may cause redundant recomputations of non-tabled subgoals that appear in the body of a looping alternative and redundant consumption of answers if the looping alternative contains more than one variant subgoal call. Parallel recomputation is harder because we do not know beforehand if a tabled alternative needs to be recomputed: a conservative approach may lose parallelism, and an optimistic approach may lead to even more redundant computation.

## 4 The Sequential Tabling Engine

Next, we review the main principles of the YapTab design (please refer to [14, 11] for more details). YapTab implements two tabling scheduling strategies, batched and local [8], and in our initial design it only considers positive programs. Tables are implemented using tries as proposed in [10]. We reconsidered decisions in the original SLG-WAM that can be a potential source of parallel overheads. Namely, YapTab considers that control of leader detection and scheduling of unconsumed answers should be performed through the consumer nodes. Hence, YapTab associates a new data structure, the *dependency frame*, to consumer nodes. In contrast, the SLG-WAM associates this control with generator nodes. We argue that managing dependencies at the level of the consumer nodes is a more intuitive approach that we can take advantage of.

The introduction of this new data structure allows us to reduce the number of extra fields in tabled choice points and to eliminate the need for a separate

completion stack. Furthermore, allocating the data-structure in a separate area simplifies the implementation of parallelism.

To benefit from the philosophy behind the dependency frame data structure, we redesigned the algorithms related with suspension, resumption and completion. We next present YapTab's main data structures and algorithms. We assume a batched scheduling strategy implementation [8] (please refer to [11] for the implementation of local scheduling).

***Generator and Consumer Nodes*** YapTab implementation stores generator nodes as standard nodes plus a pointer to the corresponding subgoal frame. In contrast to the SLG-WAM, we adjust the freeze registers by using the top of stack values kept in the consumer choice points. YapTab also implements consumer nodes as standard nodes plus a pointer to a *dependency frame*. The dependency frames are linked together to form the *dependency list of consumer nodes*. Additionally, dependency frames store information to efficiently check for completion points, replacing the need for a separate completion stack [15], as we discuss next.

***Completion and Leader Nodes*** The completion operation takes place when a generator node exhausts all alternatives and finds itself as a leader node. We designed novel algorithms to quickly determine whether a generator node is a leader node.

Our key idea is that each dependency frame holds a pointer to the presumed leader node of its SCC, and that the youngest consumer node always knows the leader for the current SCC. Hence, our leader node algorithm must always compute leader node information when first creating a new consumer node, say $\mathcal{C}$. To do so, we first hypothesize that the current leader node is $\mathcal{C}$'s generator node, say $\mathcal{G}$. Next, for all consumer nodes between $\mathcal{C}$ and $\mathcal{G}$, we check whether they depend on an older generator node. Consider that the oldest dependency is for $\mathcal{G}'$. If this is the case, then $\mathcal{G}'$ is the leader node, otherwise our hypothesis was correct and the leader is indeed $\mathcal{G}$.

Whenever we backtrack to a generator that it also the current leader node, we must check whether there are younger consumer nodes with unconsumed answers. This is implemented by going through the chain of dependency frames looking for a frame with unconsumed answers. If there is such a frame, we resume the computation to the corresponding consumer node. Otherwise, we perform completion. Completion includes **(i)** marking all the subgoals in the SCC as completed; **(ii)** deallocating all younger dependency frames; **(iii)** adjusting the freeze registers; and **(iv)** backtracking to the previous node to continue the execution.

***Answer Resolution*** Answer resolution has to be performed whenever the computation fails and is resumed at a consumer choice point. The implementation must guarantee that every answer is consumed once and just once. First, we check the table space for unconsumed answers for the subgoal at hand. If there are new answers, we load the next available answer and proceed with execution.

Otherwise, we schedule for a backtracking node. If this is the first time that backtracking from that consumer node takes place, then it is performed as usual to the previous node. Otherwise, we know that the computation has been resumed from an older generator node $\mathcal{G}$ during an unsuccessful completion operation. Therefore, backtracking must be done to the next consumer node that has unconsumed answers and that is younger than $\mathcal{G}$. If there are no such consumer nodes then backtracking must be done to the generator node $\mathcal{G}$.

## 5 The Or-Parallel Tabling Engine

The OPTYap engine is based on the YapTab engine. However, new data structures and algorithms were required to support parallel execution. Next, we describe the main design and implementation decisions.

***Memory Management*** The efficiency of a parallel system largely depends on how concurrent handling of shared data is achieved and synchronized. Page faults and memory cache misses are a major source of overhead regarding data access or update in parallel systems. OPTYap tries to avoid these overheads by adopting a page-based organization scheme to split memory among different data structures, in a way similar to Bonwick's Slab memory allocator [2].

Our experience showed that the table space is a key data area open to concurrent access operations in a parallel tabling environment. To maximize parallelism, whilst minimizing overheads, accessing and updating the table space must be carefully controlled. Read/write locks are the ideal implementation scheme for this purpose. OPTYap implements four alternative locking schemes to deal with concurrent accesses to the table data structures. Our results suggested that concurrent table access is best handled by schemes that lock table data only when writing to the table is likely.

***Leader Nodes*** Or-parallel systems execute alternatives early. As a result, it is possible that generators will execute earlier, and in a different branch than in sequential execution. In the worst case, different workers may execute the generator and the consumer goals. Workers may have consumer nodes while not having the corresponding generators in their branches. Or, the owner of a generator node may have consumers being executed by several different workers. This may induce complex dependencies between workers, hence requiring a more elaborate completion operation that may involve branches created by several workers.

OPTYap allows completion to take place at any node, not only at generator nodes. In order to allow a very flexible completion algorithm we introduce a new concept, the *generator dependency node* (or *GDN*). Its purpose is to signal the nodes that are candidates to be leader nodes, therefore representing a similar role as that of the generator nodes for sequential tabling. The GDN is calculated whenever a new consumer node, say $\mathcal{C}$, is allocated. It is defined as the youngest node $\mathcal{D}$ on the current branch of $\mathcal{C}$, that is an ancestor of the generator node

$\mathcal{G}$ for $\mathcal{C}$. Figure 1 presents three different situations that better illustrate the GDN concept. $\mathcal{WG}$ is the worker that allocated the generator node $\mathcal{G}$, $\mathcal{WC}$ is the worker that is allocating a consumer node $\mathcal{C}$, and the node pointed by the black arrow is the GDN for the new consumer.

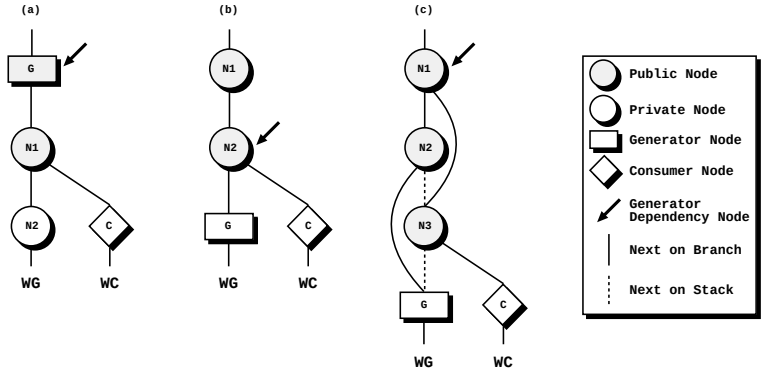


**Fig. 1.** Spotting the generator dependency node.

In situation **(a)**, the generator node $\mathcal{G}$ is on $\mathcal{C}$'s branch, and thus, $\mathcal{G}$ is the GDN. In situation **(b)**, nodes $\mathcal{N}_1$ and $\mathcal{N}_2$ are on $\mathcal{C}$'s branch, and both contain a branch leading to $\mathcal{G}$. As $\mathcal{N}_2$ is the youngest node of both, it is the GDN. In situation **(c)**, $\mathcal{N}_1$ is the unique node that belongs to $\mathcal{C}$'s branch and that also contains $\mathcal{G}$ in a branch below. $\mathcal{N}_2$ contains $\mathcal{G}$ in a branch below, but it is not on $\mathcal{C}$'s branch, while $\mathcal{N}_3$ is on $\mathcal{C}$'s branch, but it does not contain $\mathcal{G}$ in a branch below. Therefore, $\mathcal{N}_1$ is the GDN. Notice that in both cases **(b)** and **(c)** the GDN can be a generator, a consumer or an interior node.

The procedure to compute the leader node information when allocating a dependency frame for a new consumer node now hypothesizes that the leader node for the consumer node at hand is its GDN, and not its generator node.

***The Control Flow*** OPTYap's execution control mainly flows through four procedures. The process of completely evaluating SCCs is accomplished by the `completion()` and `answer_resolution()` procedures, while parallel synchronization is achieved by the `getwork()` and `scheduler()` procedures. Here we focus on the flow of control in engine mode, that is on the `completion()`, `answer_resolution()` and `getwork()` procedures, and discuss scheduling later. Figure 2 presents a general overview of how control flows between the three procedures and how it flows within each procedure.

*Public Completion* Different paths may be followed when a worker $\mathcal{W}$ reaches a leader node for a SCC $\mathcal{S}$. The simplest case is when the node is private. In this case, we proceed as for sequential tabling. Otherwise, the node is public, and there *may* exist dependencies on branches explored by other workers. Therefore,

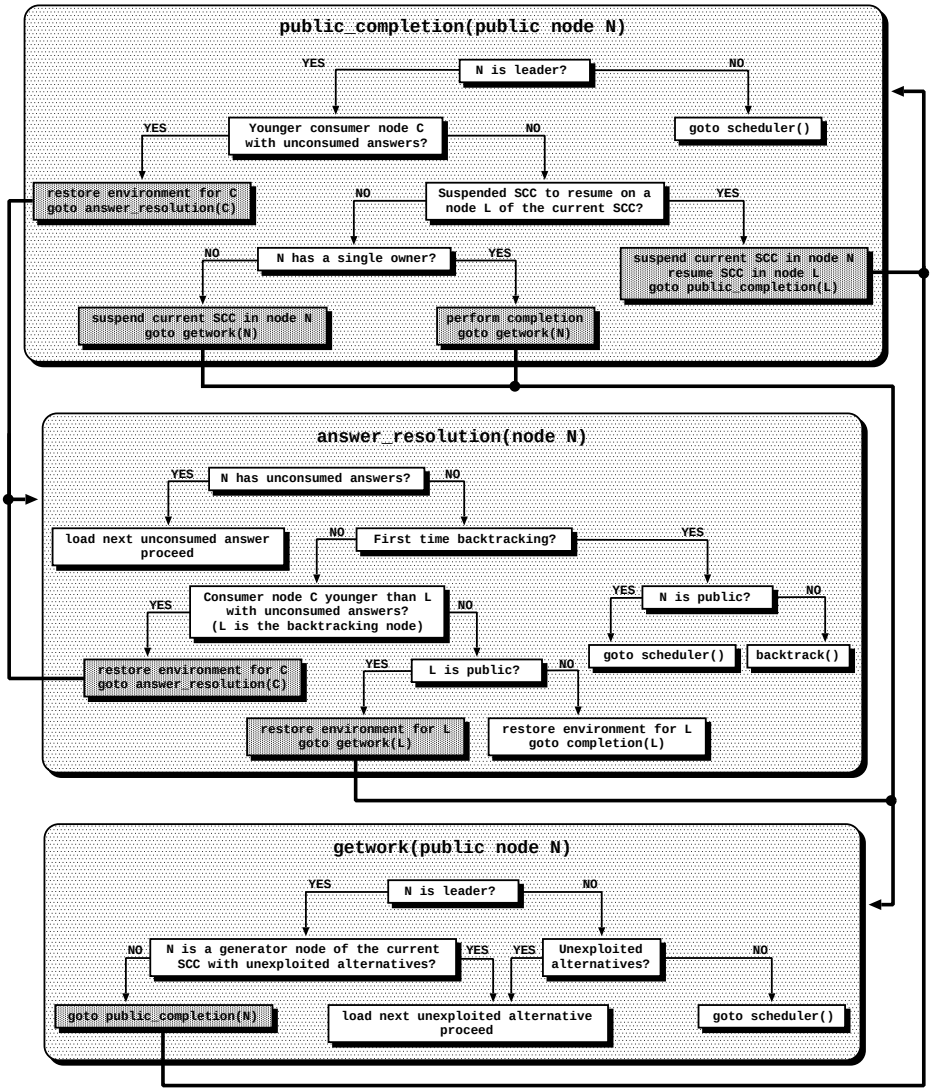


**Fig. 2.** The flow of control in a parallel tabled evaluation.

even when all younger consumer nodes on $\mathcal{W}$'s stacks do not have unconsumed answers, completion *cannot* be performed. The reason for this is that the other workers can still influence $\mathcal{S}$. For instance, these workers may find new answers for a consumer node in $\mathcal{S}$, in which case the consumer must be resumed to consume the new answers. As a result, in order to allow $\mathcal{W}$ to continue execution it becomes necessary to *suspend the SCC* at hand.

Suspending in this context is obviously different from suspending consumer nodes. Consumer nodes are suspended due to tabled evaluation. SCCs are sus-

pended due to or-parallel execution. Suspending a SCC includes saving the SCC's stacks to a proper space, leaving in the leader node a reference to where the stacks were saved, and readjusting the freeze registers and the stack and frame pointers. If the worker did not suspend the SCC, hence not saving the stacks, any future sharing work operation might damage the SCC's stacks and therefore make delayed completion unworkable.

To deal with the new particularities arising with concurrent evaluation a novel completion procedure, `public_completion()`, implements completion detection for public leader nodes. As for private nodes, whenever a public node finds that it is a leader, it starts to check for younger consumer nodes with unconsumed answers. If there is such a node, we resume the computation to it. Otherwise, it checks for suspended SCCs in the scope of its SCC. A suspended SCC should be resumed if it contains consumer nodes with unconsumed answers. To resume a suspended SCC a worker needs to copy the saved stacks to the correct position in its own stacks, and thus, it has to suspend its current SCC first.

We thus adopted the strategy of resuming suspended SCCs *only when the worker finds itself at a leader node*, since this is a decision point where the worker either completes or suspends the current SCC. Hence, if the worker resumes a suspended SCC it does not introduce further dependencies. This is not the case if the worker would resume a suspended SCC $\mathcal{R}$ as soon as it reached the node where it had suspended. In that situation, the worker would have to suspend its current SCC $\mathcal{S}$, and after resuming $\mathcal{R}$ it would probably have to also resume $\mathcal{S}$ to continue its execution. A first disadvantage is that the worker would have to make more suspensions and resumptions. Moreover, if we resume earlier, $\mathcal{R}$ may include consumer nodes with unconsumed answers that are common with $\mathcal{S}$. More importantly, suspending in non-leader nodes leads to further complexity. Answers can be found in upper branches for suspensions made in lower nodes, and this can be very difficult to manage.

A SCC $\mathcal{S}$ is completely evaluated when **(i)** there are no unconsumed answers in any consumer node in its scope, that is, in any consumer node belonging to $\mathcal{S}$ or in any consumer node within a SCC suspended in a node belonging to $\mathcal{S}$; and **(ii)** there is only a single worker owning its leader node $\mathcal{L}$. We say that a worker *owns* a node $\mathcal{N}$ when it holds $\mathcal{N}$ on its stacks (this is true even if $\mathcal{N}$ is not the worker's current branch). Completing a SCC includes **(i)** marking all dependent subgoals as complete; **(ii)** releasing the frames belonging to the complete branches, including the branches in suspended SCCs; **(iii)** releasing the frozen stacks and the memory space used to hold the stacks from suspended SCCs; and **(iv)** readjusting the freeze registers and the whole set of stack and frame pointers.

Our public completion algorithm has two major advantages. One is that the worker checking for completion determines if its current SCC is completely evaluated or not without requiring any explicit communication or synchronization with other workers. The other is that it uses the SCC as the unit for suspension. This latter advantage is very important since it simplifies the management of dependencies arising from branches not on stack. A leader node determines the

position from where dependencies may exist in younger branches. As a suspension unit includes the whole SCC and suspension only occurs in leader node positions, we can simply use the leader node to represent the whole scope of a suspended SCC, and therefore simplify its management.

*Answer Resolution* The answer resolution operation for the parallel environment essentially uses the same algorithm as previously described for private nodes.

*Getwork* The last flow control procedure. It contributes to the progress of a parallel tabled evaluation by moving to effective work. The usual way to execute `getwork()` is through failure to the youngest public node on the current branch. We can distinguish two blocks of code in the `getwork()` procedure. The first block detects completion points and therefore makes the computation flow to the `public_completion()` procedure. The second block corresponds to or-parallel execution. It synchronizes to check for available alternatives and executes the next one, if any. Otherwise, it invokes the scheduler.

The `getwork()` procedure detects a completion point when $\mathcal{N}$ is the leader node pointed by the top dependency frame. The exception is if $\mathcal{N}$ is itself a generator node for a consumer node within the current SCC and it contains unexploited alternatives. In such cases, the current SCC is not fully exploited. Hence, we should exploit first the available alternatives, and only then invoke completion.

***Scheduling Work*** Scheduling work is the scheduler's task. It is about efficiently distributing the available work for exploitation between the running workers. In a parallel tabling environment we have the extra constraint of keeping the correctness of sequential tabling semantics. A worker enters in scheduling mode when it runs out of work and returns to execution whenever a new piece of unexploited work is assigned to it by the scheduler.

The scheduler for the OPTYap engine is mainly based on YapOr's scheduler. All the scheduler strategies implemented for YapOr were used in OPTYap. However, extensions were introduced in order to preserve the correctness of tabling semantics. These extensions allow support for leader nodes, frozen stack segments, and suspended SCCs. The OPTYap model was designed to enclose the computation within a SCC until the SCC was suspended or completely evaluated. Thus, OPTYap introduces the constraint that the *computation cannot flow outside the current SCC, and workers cannot be scheduled to execute at nodes older than their current leader node.* Therefore, when scheduling for the nearest node with unexploited alternatives, if it is found that the current leader node is younger than the potential nearest node with unexploited alternatives, then the current leader node is the node scheduled to proceed with the evaluation.

*Moving In the Tree* The next case is when the process above does not return any node to proceed execution. The scheduler then starts searching for busy workers that can be requested for work. If such a worker $\mathcal{B}$ is found, then the requesting worker moves up to the lowest node that is common to $\mathcal{B}$, in order

to become partially consistent with part of $\mathcal{B}$. Otherwise, no busy worker was found, and the scheduler moves the idle worker to a better position in the search tree. Therefore, we can enumerate three different situations for a worker to move up to a node $\mathcal{N}$: **(i)** $\mathcal{N}$ is the nearest node with unexploited alternatives; **(ii)** $\mathcal{N}$ is the lowest node common with the busy worker we found; or **(iii)** $\mathcal{N}$ corresponds to a better position in the search tree.

The process of moving up in the search tree from a current node $\mathcal{N}_0$ to a target node $\mathcal{N}_f$ is implemented by the `move_up_one_node()` procedure. This procedure is invoked for each node that has to be traversed until reaching $\mathcal{N}_f$. The presence of frozen stack segments or the presence of suspended SCCs in the nodes being traversed influences and can even abort the usual moving up process.

Assume that the idle worker $\mathcal{W}$ is currently positioned at $\mathcal{N}_i$ and that it wants to move up one node. Initially, the procedure checks for frozen nodes on the stack to infer whether $\mathcal{W}$ is moving within the SCC. If so, $\mathcal{W}$ is simply deleted from member of $\mathcal{N}_i$. The interesting case is when $\mathcal{W}$ is not within a SCC. If $\mathcal{N}_i$ holds a suspended SCC, then $\mathcal{W}$ can safely resume it. If resumption does not take place, the procedure proceeds to check whether $\mathcal{N}_i$ is a consumer node. Being this the case, $\mathcal{W}$ is deleted from member of $\mathcal{N}_i$ and if $\mathcal{W}$ is the unique owner of $\mathcal{N}_i$ then the suspended SCCs in $\mathcal{N}_i$ can be completed. Completion can be safely performed over the suspended SCCs in $\mathcal{N}_i$ not only because the SCCs are completely evaluated, as none was previously resumed, but also because no more dependencies exist, as there are no more branches below $\mathcal{N}_i$. The reasons given to complete the suspended SCCs in $\mathcal{N}_i$ hold even if $\mathcal{N}_i$ is not a consumer node, as long as $\mathcal{W}$ is the unique owner of $\mathcal{N}_i$. In such case, if $\mathcal{N}_i$ is a generator node then its correspondent subgoal can be also marked as completed. Otherwise, $\mathcal{W}$ is simply deleted from being member and owner of $\mathcal{N}_i$.

## 6 Initial Performance Evaluation

The environment for our experiments consists of a shared memory parallel machine, a 200 MHz PentiumPro with 4 processors, 128 MBytes of main memory, 256 KBytes of cache and running the linux-2.2.12 kernel. The machine was otherwise idle while benchmarking.

YapOr, YapTab and OPTYap are based on Yap's 4.2.1 engine. Note that sequential execution would be somewhat better with more recent Yap engines. We used the same compilation flags for Yap, YapOr, YapTab and OPTYap. Regarding XSB Prolog, we used version 2.3 with the default configuration and the default execution parameters (chat engine and batched scheduling).

***Non-Tabled Benchmarks*** To put the performance results in perspective we first use a common set of non-tabled benchmark programs to evaluate how the original Yap Prolog engine compares against the several Yap extensions and against the most well-known tabling engine, XSB Prolog. The benchmarks include the n-queens problem, the puzzle and cubes problems from Evan Tick's

book, an hamiltonian graph problem and a naïve sort algorithm. All benchmarks find all solutions for the problem.

Table 1 shows the base running times, in milliseconds, for Yap, YapOr, YapTab, OPTYap and XSB for the set of non-tabled benchmarks. In parentheses, it shows the overhead over the Yap running times. The results indicate that YapOr, YapTab and OPTYap introduce, on average, an overhead of about 6%, 8% and 12% respectively over standard Yap. Regarding XSB, the results show that, on average, XSB is 1.9 times slower than Yap, a result mainly due to the faster Yap engine.

| **Program** | **Yap** | **YapOr** | **YapTab** | **OPTYap** | **XSB** |
|---|---|---|---|---|---|
| 9-queens | 584 | 604(1.03) | 605(1.04) | 626(1.07) | 1100(1.88) |
| cubes | 170 | 170(1.00) | 173(1.02) | 175(1.03) | 329(1.94) |
| ham | 371 | 402(1.08) | 399(1.08) | 432(1.16) | 659(1.78) |
| nsort | 310 | 330(1.06) | 328(1.06) | 354(1.14) | 629(2.03) |
| puzzle | 1633 | 1818(1.11) | 1934(1.18) | 1950(1.19) | 3059(1.87) |
| *Average* | | (1.06) | (1.08) | (1.12) | (1.90) |

**Table 1.** Running times on non-tabled programs.

YapOr overheads result from handling the work load register and from testing operations that **(i)** verify whether a node is shared or private, **(ii)** check for sharing requests, and **(iii)** check for backtracking messages due to cut operations. On the other hand, YapTab overheads are due to the handling of the freeze registers and support of the forward trail. OPTYap overheads result from both.

Since OPTYap is based on the same environment model as the one used by YapOr, we then compare OPTYap's parallel performance with that of YapOr. Table 2 shows the speedups relative to the single worker case for YapOr and OPTYap with 2, 3 and 4 workers. Each speedup corresponds to the best execution time obtained in a set of 3 runs. The results show that OPTYap maintains YapOr's behavior in exploiting or-parallelism in non-tabled programs, despite that it includes all the machinery required to support tabled programs.

| | **YapOr** | | | **OPTYap** | | |
|---|---|---|---|---|---|---|
| **Program** | **2** | **3** | **4** | **2** | **3** | **4** |
| 9-queens | 1.99 | 2.99 | 3.94 | 2.00 | 2.99 | 3.96 |
| cubes | 2.00 | 2.98 | 3.95 | 1.98 | 2.96 | 3.97 |
| ham | 2.00 | 2.95 | 3.90 | 1.97 | 2.93 | 3.78 |
| nsort | 1.97 | 2.92 | 3.83 | 1.97 | 2.92 | 3.80 |
| puzzle | 2.02 | 3.03 | 4.02 | 1.98 | 2.97 | 3.94 |
| *Average* | 2.00 | 2.97 | 3.93 | 1.98 | 2.95 | 3.89 |

**Table 2.** Speedups for YapOr and OPTYap on non-tabled programs.

***Tabled Benchmarks*** We then use a set of tabled benchmark programs to measure the performance of the tabling engines in discussion. The benchmarks

include two transition systems from XMC specs[1], a same generation problem for a 24x24x2 data cylinder, and two path problems that find the transitive closure of different graph topologies. All benchmarks find all the solutions for the problem.

Table 3 shows the base running times, in milliseconds, for YapTab, OPTYap and XSB for the set of tabled benchmarks. In parentheses, it shows the overhead over the YapTab running times. The results indicate that OPTYap introduce, on average, an overhead of about 17% over YapTab for tabled programs, which is much worse than the overhead of 5% for non-tabled programs. The difference results from locking requests to handle the data structures introduced by tabling. Locks are require to insert new trie nodes into the table space, and to update subgoal and dependency frame pointers to tabled answers. We observed that the benchmarks that deal with more tabled answers per time unit are the ones that perform more locking operations and in consequence introduce further overheads.

| **Program** | **YapTab** | **OPTYap** | **XSB** |
|---|---|---|---|
| xmc-sieve | 2851 | 3226(1.13) | 3560(1.25) |
| xmc-iproto | 2438 | 2736(1.22) | 4481(1.84) |
| same-gen | 16598 | 17034(1.03) | 25390(1.82) |
| path-grid | 1069 | 1240(1.16) | 3610(3.38) |
| path-chain | 102 | 136(1.33) | 271(2.66) |
| *Average* | | (1.17) | (2.19) |

**Table 3.** Running times on tabled programs.

Regarding XSB, the results show that, on average, YapTab is slightly more than twice as fast as XSB, surprisingly a better result than for non-tabled benchmarks. In particular, XSB shows the worst behavior for the two programs that are more table intensive. We believe that the XSB performance may be caused by overheads in their tabling implementation. XSB must support negated literals, and also has recently been extended to support attributed variables and especially subsumption.

***Parallel Tabled Benchmarks*** To assess the performance of OPTYap when running the tabled programs in parallel, we ran OPTYap for the same set of tabled programs with varying number of workers. Table 4 shows the speedups relative to the single worker case for OPTYap with 2, 3 and 4 workers. Each speedup corresponds to the best execution time obtained in a set of 3 runs. The table is divided in two blocks: the upper block groups the benchmarks that showed potential for parallel execution, whilst the lower block includes the benchmark that do not show any gains when run in parallel.

Globally, our results show quite good speedups for the upper block programs, especially considering that the execution times were obtained in a multiprocess environment. In particular, *xmc-sieve* achieves linear speedups up to 4 workers.

[1] We are thankful to C.R. Ramakrishnan for providing us these benchmarks.

| Program | Number of Workers | | |
|---|---|---|---|
| | 2 | 3 | 4 |
| xmc-sieve | 2.00 | 3.00 | 3.99 |
| xmc-iproto | 1.90 | 2.78 | 3.64 |
| same-gen | 2.04 | 2.84 | 3.86 |
| path-grid | 1.82 | 2.54 | 3.10 |
| *Average* | 1.94 | 2.79 | 3.65 |
| path-chain | 0.92 | 0.86 | 0.78 |

**Table 4.** Speedups for OPTYap on tabled programs.

The *same-gen* benchmark presents also excellent results up to 4 workers and *xmc-iproto* and *path-grid* show a slightly slowdown with the increase in the number of workers. On the other hand, the `path-chain` benchmark does not show any speedup at all.

Through experimentation, we observed that workers are busy for more than 95% of the execution time, even for 4 workers. In general, slowdowns are not caused because workers became idle and start searching for work, as usually happens with parallel execution of non-tabled programs. Here the problem seems more complex: workers do have available work, but there is a lot of contention to access that work.

Closer analysis suggested that there are two main reasons that constraint speedups. One relates with massive table access to insert and consume answers. As trie structures are a compact data structure, the presence of massive table access increases the number of contention points. The other relates with the sequencing in the order that answers are found. There are answers that can only be found when other answers are also found, and the process of finding such answers cannot be anticipated. This incurs in high overheads related with SCC suspensions and resumptions.

## 7 Conclusions

In this paper we have presented the design and implementation of OPTYap. To the best of our knowledge, OPTYap is the first parallel tabling engine for logic programming systems. OPTYap extends the Yap Prolog system both with the SLG-WAM, initially implemented for XSB Prolog, and with environment copying, initially implemented in the Muse or-parallel system.

First results show that OPTYap introduces low overheads for sequential execution, and that it compares favorably with current versions of XSB. Moreover, the results showed that OPTYap maintains YapOr's effective speedups in exploiting or-parallelism in non-tabled programs. For parallel execution of tabled programs, OPTYap showed linear speedups for a well known application of XSB, and quite good results globally. These results emphasize our belief that tabling and parallelism are a very good match.

On the other hand, there are tabled programs where OPTYap may not speedup up execution. Parallel execution of tabled programs may have different

characteristics than traditional or-parallel programs. In general, tabling tends to decrease the height of the search tree, whilst increasing its breadth. We therefore believe that improvements in scheduling and on concurrent access to tries may be fundamental for scalable performance. We plan to investigate this issue further, also by studying more programs.

## References


1. K. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *Journal of Parallel Programming*, 19(2):129–162, 1990.
2. Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer 1994*, pages 87–98, 1994.
3. W. Chen and D. S. Warren. Query Evaluation under the Well Founded Semantics. In *Proceedings of PODS*, pages 168–179, 1993.
4. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.
5. B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *Proceedings of PLILP*, number 1490 in LNCS, pages 21–35. Springer-Verlag, 1998.
6. B. Demoen and K. Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. In *Proceedings of PADL*, number 1551 in LNCS, pages 106–121. Springer-Verlag, 1999.
7. J. Freire, R. Hu, T. Swift, and D. S. Warren. Exploiting Parallelism in Tabled Evaluations. In *Proceedings of PLILP*, pages 115–132. Springer-Verlag, 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*, pages 243–258. Springer-Verlag, 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*, 2000.
10. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *Proceedings of ICLP*, pages 687–711. The MIT Press, 1995.
11. R. Rocha. *On Applying Or-Parallelism and Tabling to Logic Programs.* PhD thesis, Computer Science Department, University of Porto, 2001.
12. R. Rocha, F. Silva, and V. Santos Costa. Or-Parallelism within Tabling. In *Proceedings of PADL*, number 1551 in LNCS, pages 137–151. Springer-Verlag, 1999.
13. R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Proceedings of EPIA*, number 1695 in LNAI, pages 178–192. Springer-Verlag, 1999.
14. R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Proceedings of TAPD*, pages 77–87, 2000.
15. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *Journal of ACM Transactions on Programming Languages and Systems*, 1998.
16. Vítor Santos Costa. Optimising Bytecode Emulation for Prolog. In *Proceedings of PPDP*, number 1702 in LNCS, pages 261–267. Springer-Verlag, 1999.
17. David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.
18. Neng-Fa Zhou. Implementation of a Linear Tabling Mechanism. In *Proceedings of PADL*, number 1753 in LNCS, pages 109–123. Springer Verlag, 2000.