IJPP manuscript No.
(will be inserted by the editor)

Simpler is Faster: Multi-Dimensional Lock-Free Arrays for Multithreaded Mode-Directed Tabling in Prolog

Miguel Areias · Ricardo Rocha

Received: date / Accepted: date

Abstract This work proposes a new design for the supporting data structures used to implement multithreaded tabling in Prolog systems. Tabling is a implementation technique that improves the expressiveness of traditional Prolog systems in dealing with recursion and redundant computations. Mode-directed tabling is an extension to the tabling technique that supports the definition of alternative criteria for specifying how answers are aggregated, being thus very suitable for problems where the goal is to dynamically calculate optimal or selective answers. In this work, we leverage the intrinsic potential that mode-directed tabling has to express dynamic programming problems, by creating a new design that improves the representation of multi-dimensional arrays in the context of multithreaded tabling. To do so, we introduce a new mode for indexing arguments in mode-directed tabled evaluations, named *dim*, where each *dim* argument features a uni-dimensional lock-free array. Experimental results using well-known dynamic programming problems on a 32-core machine show that the new design introduces less overheads and clearly improves the execution time for sequential and multithreaded tabled evaluations.

Keywords Prolog · Tabling · Dynamic Programming · Lock-Freedom

This work was funded by the ERDF through Project 9471-RIDTI and by the NORTE 2020 Programme under the PORTUGAL 2020 Partnership Agreement, as part of project NanoSTIMA (NORTE-01-0145-FEDER-000016), and through the COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the FCT as part of project UID/EEA/50014/2013. Miguel Areias was funded by the FCT grant SFRH/BPD/108018/2015.

Miguel Areias
CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
E-mail: miguel-areias@dcc.fc.up.pt

Ricardo Rocha
CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
E-mail: ricroc@dcc.fc.up.pt

1 Introduction

Dynamic programming [3] is a general recursive strategy that consists in dividing a problem in simple sub-problems that, often, are the same. The idea behind dynamic programming is to reduce the number of computations: once an answer to a given sub-problem has been computed, it is memoized and the next time the same answer is needed, it is simply looked up. Dynamic programming is especially useful when the number of overlapping sub-problems grows exponentially as a function of the size of the input, but their size is polynomial when viewed as a set.

Tabling (or memoing) [4] is a kind of dynamic programming implementation technique that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant sub-computations. Tabling is a refinement of Prolog's default resolution that stems from one simple idea: save intermediate answers for current computations in an appropriate data area, called the *table space*, so that they can be reused when a *similar computation* appears during the resolution process. Tabled evaluation can reduce the search space, avoid looping and have better termination properties than Prolog's default resolution. Work on tabling proved its viability for application areas such as deductive databases [16], inductive logic programming [14], knowledge based systems [21], model checking [12], parsing [8], program analysis [5], reasoning in the semantic Web [23], among many others. Currently, the tabling technique is widely available in systems like B-Prolog, Ciao, Mercury, Picat, SWI-Prolog, XSB Prolog and YAP Prolog. Mode-directed tabling [6] is an extension to the tabling technique that supports the definition of alternative criteria, or *modes*, for specifying how answers are inserted into the table space. The key idea is to define the terms of the arguments that define sub-computations to be considered for similarity[1] checking (the index arguments) and define additionally how variant answers of those sub-computations should be tabled (or stored) regarding the remaining arguments (the output arguments) [6]. Mode-directed tabling is thus very suitable for problems where the goal is to dynamically calculate optimal or selective answers as new results arrive.

Multithreading in Prolog is the ability to concurrently perform computations, in which each computation runs independently but shares the program clauses. When multithreading is combined with tabling, we have the best of both worlds, since we can exploit the combination of higher procedural control with higher declarative semantics. In a multithreaded tabling system, we have the extra problem of ensuring the correctness and completeness of the concurrent answers found and stored in the tables. Thus, despite the availability of threads and tabling in a Prolog system, supporting both features simultaneously implies complex ties to the underlying engine. To the best of our knowledge, XSB Prolog [10] and YAP Prolog [1] are the only systems that support the combination of tabling with multithreading.

[1] Two terms are considered to be similar if they are the same up to variable renaming.

In this work, we leverage the intrinsic potential that mode-directed tabling has to express dynamic programming problems, by creating a new design that improves the representation of multi-dimensional arrays in the context of multithreaded tabling. To do so, we introduce a new mode for indexing arguments in mode-directed tabling, named *dim*, where each *dim* argument features a uni-dimensional lock-free array. This functionality allows users to explicitly define arguments specially aimed for a fast evaluation of dynamic programming problems with single solutions for multiple integer dimensions, like the ones which calculate the maximum/minimum value of a sub-computation. Our focus on multi-dimensional arrays emerged because, several of the proposals that can be found in the literature to parallelize dynamic programming problems, are based on a careful analysis of the sequential algorithm, in order to find the best way to minimize the data dependencies in the supporting data structure for memoization, often a multi-dimensional array [15,9].

To the best of our knowledge, this is the first work on multithreaded tabling that offers such a design. We will focus our discussion on YAP's specific implementation[2], but our proposal can be generalized and applied to other tabling systems. Experimental results, on a 32-core AMD machine, show that our proposal is able to improve greatly the execution time of well-known dynamic programming problems by taking advantage of the new multi-dimensional and lock-free design. The new design introduces less overheads and clearly improves the execution time for sequential and multithreaded execution. In particular, for multithreaded execution up to 32 threads, the new design showed to be able to maintain or achieve slightly better speedups despite its base execution times (with one thread) be 1.5 to 2.5 times faster than the previous design. With the results obtained, we expect that multithreaded tabling can be seen as a relevant member within the general ecosystem of concurrent/parallel environments for the evaluation of dynamic programming problems.

The remainder of the paper is organized as follows. First, we briefly introduce some background about tabling in Prolog systems, mode-directed tabling and multithreaded tabling. Next, we introduce our new table space design. Then, we describe in detail the key algorithms that support the implementation and discuss their correctness. Finally, we present experimental results and we end by outlining some conclusions.

2 Background

Dynamic programming can be implemented using either a *bottom-up* or a *top-down* approach. Bottom-up approaches start from the base sub-problems and recursively compute the next level sub-problems until reaching the answer to the given problem. On the other hand, top-down approaches start from the given problem and use recursion to subdivide a problem into sub-problems until reaching the base sub-problems. Answers to previously computed sub-

[2] Available at `https://github.com/miar/yap-6.3`.

problems are reused rather than being recomputed. An advantage of the top-down approach is that it might not need to compute all possible sub-problems. However, dynamic programming has some limitations, such as, the *curse of dimensionality* [3] which might occur in problems with high-dimensional spaces (often with hundreds or thousands of dimensions) where the volume of space is so high that the available data becomes sparse, thus preventing common data organization strategies from being efficient. In this work, we focus on problems with low-dimensional integer spaces, such as the Knapsack and the Longest Common Subsequence (LCS) problems.

2.1 Tabling in Prolog Systems

The key idea of tabling is to have a special type of call, named *tabled call*, which is used to minimize the evaluation of the search space in the same fashion as the standard dynamic programming techniques. To do so, tabling uses an auxiliary data space, called the *table space*, to keep track of the subgoal calls in evaluation and store, for each subgoal, the *set of answers* which are found during program's evaluation. Whenever a similar subgoal call appears, it is resolved by consuming answers from table space instead of executing the program clauses. During this process, as further new answers are found, they are added to their tables and later returned to all similar calls. To control the execution flow, tabling uses two types of special data structures: (i) *generator nodes*, which correspond to first calls to a tabled subgoal, are used to generate the answers for the call; and (ii) *consumer nodes*, which correspond to similar calls to a tabled subgoal, are used to consume the answers found for the corresponding generator call.

With these requirements, the design of the table space is critical to achieve an efficient implementation. YAP uses *tries* which is regarded as a very efficient way to implement the table space [13]. Tries are trees in which common prefixes are represented only once. YAP 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 answers for the calls. Figure 1 shows YAP's default table space organization. At the entry point we have the *table entry* data structure. This structure is allocated when

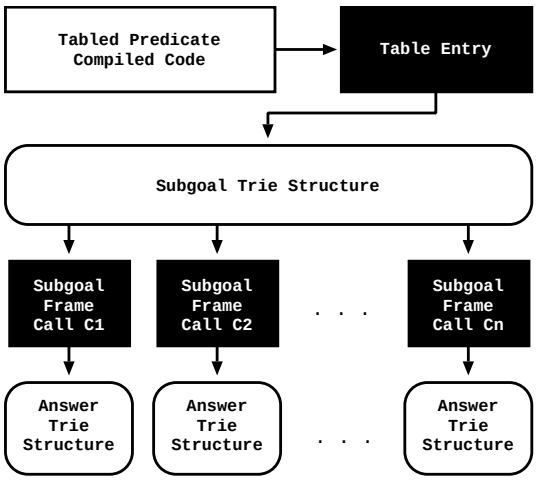

Fig. 1 YAP's default table space organization

a predicate is being compiled, thus guaranteeing that all calls to the predicate will access the table space starting from the same point. Below the table entry, we have the *subgoal trie structure*. Each different tabled call corresponds to a unique path through the subgoal trie structure, always starting from the table entry, passing by several subgoal trie data units, the *subgoal trie nodes*, and reaching a leaf data structure, the *subgoal frame*. The subgoal frame stores additional information about the subgoal and acts like an entry point to the *answer trie structure*. Each unique path through the answer trie data units, the *answer trie nodes*, corresponds to a different tabled answer to the entry subgoal. At the engine level, generator and consumer nodes access the table space by keeping a reference to the corresponding subgoal frame.

2.2 Mode-Directed Tabling

In traditional tabling, 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 of any answer that is already in the table space, then it is always considered for insertion. Therefore, traditional tabling is very good for problems that require storing all answers. However, with dynamic programming, usually, the goal is to dynamically calculate optimal or selective answers as new results arrive.

Mode-directed tabling [6] is an extension to the tabling technique that supports the definition of *modes* for specifying how answers are inserted into the table space. Within 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. The idea is to define the arguments to be considered for similarity checking (the index arguments) and how variant answers should be tabled regarding the remaining arguments (the output arguments). Implementations of mode-directed tabling are currently available in B-Prolog [22] and YAP Prolog [17]. A restricted form of mode-directed tabling can also be reproduced in XSB Prolog by using *answer subsumption* [19]. In YAP, index arguments are represented with mode *index*, while arguments with modes *first*, *last*, *min*, *max*, *sum* and *all* represent output arguments. When an answer is generated, the system tables the answer only if it is *preferable*, accordingly to the meaning of the output arguments, than some existing variant answer.

Figure 2 shows the Prolog code that implements a generic version of the Knapsack problem using mode-directed tabling. The table directive declares that predicate *ks* with arity 3 (or $ks/3$ for short) is to be tabled using modes $(index, index, max)$, meaning that the third argument (the profit) should store only the maximal answers for the first two arguments (the index of the number of items being considered and the knapsack's capacity). The code that follows implements a recursive top-down definition of the Knapsack problem. The first clause is the base case and defines that the empty set is a solution with profit 0. The second clause excludes the current item from the solution set and the third includes the current item in the solution if its inclusion does not overcome the current capacity of the knapsack.

```
% table declaration
:- table ks(index, index, max).

% base case
ks(0, Capacity, 0).
% exclude item N from the knapsack
ks(N, Capacity, Profit) :-
  N > 0, M is N - 1, ks(M, Capacity, Profit).
% include item N in the knapsack
ks(N, Capacity, Profit) :-
  N > 0, item(N, Weight_N, Profit_N),
  Capacity_M is Capacity - Weight_N, Capacity_M >= 0, M is N - 1,
  ks(M, Capacity_M, Profit_M), Profit is Profit_N + Profit_M.
```

Fig. 2 The Knapsack problem with mode-directed tabling

2.3 Multithreaded Tabling

YAP follows a SWI-Prolog compatible multithreading implementation [20], where each Prolog thread is an operating system native thread running a Prolog engine. After being started from a goal, a thread evaluates the goal just like a regular Prolog evaluation. At the engine level, each thread has its own execution stacks, with generator and consumer nodes, and only shares the code area where predicates, records, flags and other global data are stored.

For tabled evaluation, a thread views its tables as private but, at the engine level, parts of the table space can be shared among threads. One such approach is the *subgoal sharing design with shared answers of completed subgoals* [2]. The idea is as follows. The subgoal trie structures are shared among all threads and the leaf data structures representing each tabled subgoal call C_i, instead of pointing to a single subgoal frame, point to a chain of private subgoal frames, one per thread that is evaluating the call C_i. The answers for call C_i for each thread are then also stored in an answer trie structure private to each thread. Later, when the first subgoal frame is completed, i.e., when a thread has found the full set of answers for it, the subgoal frame is marked as completed and put in the beginning of the chain of private subgoal frames.

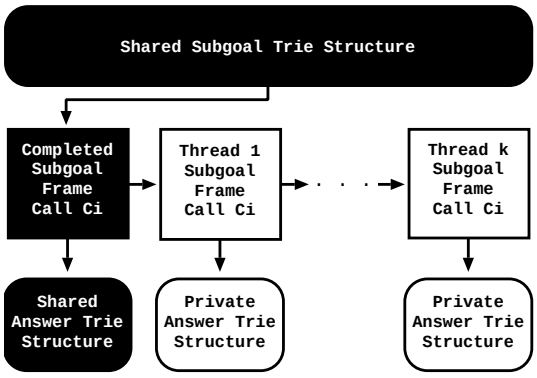

Fig. 3 Subgoal sharing design with shared answers of completed subgoals

Figure 3 illustrates this scenario in the context of a tabled subgoal call C_i. Whenever a thread calls a new tabled subgoal call, first it searches the table

space looking if any other thread has already computed the full set of answers for that call, i.e., it looks for a completed subgoal frame in the beginning of the chain. If so, it reuses the available answers, thus avoiding recomputing them from scratch. Otherwise, it computes the call itself in a private fashion. Several threads can work on the same subgoal call simultaneously. The first thread completing a call shares the results by making them publicly available.

3 Multi-Dimensional Lock-Free Table Space Design

In this work, we propose a new table space design which supports the efficient handling of multi-dimensional arrays in the context of multithreaded mode-directed tabling. The new design replaces the usage of the subgoal and answer trie data structures with uniquely identifiable bucket entries. In the new design, the multi-dimensional array represents the set of possible different calls for the tabled predicate at hand and each bucket entry in the array represents a particular subgoal call SC. Each bucket entry includes two fields: (i) one field stores the entry point for the chain of subgoal frames for SC (one frame per thread that is evaluating SC); and (ii) a second field stores the answer which represents the current aggregated answer for SC. In the current version, we support the aggregator modes *min*, *max* and *sum*.

To take advantage of the new design, we propose a new mode for indexing arguments in mode-directed tabled evaluations, named *dim*, where each *dim* mode features one dimension in the multi-dimensional array representing the tabled predicate at hand. The *dim* mode should be used with an argument representing the size of the dimension, i.e., something like *dim(N)*, where N represents the interval of integer numbers (between 0 and $N-1$) which can appear in the calls to the predicate during evaluation.

Figure 4 illustrates the new design in the context of predicate *ks/3* from Fig. 2, but now adapted to take advantage of the *dim* mode declarations. In the example, predicate *ks/3* is assumed to have been declared as *table ks(dim(X),dim(Y),max)*, where X and Y are specific integer values. As the previous design, the entry point is the table entry data structure, which for mode-directed tabling includes a pointer to a *modes data structure* storing the modes declared for the predicate. Since *ks/3* was declared using two *dim* modes, the table entry then points to a two-dimensional array with $X * Y$ bucket entries. Additionally, Fig. 4 illustrates a configuration where two subgoal calls are in evaluation, $ks(0, Y-1, VAR_0)$ and $ks(X-1, Y-1, VAR_0)$ with the aggregated answers Ans_1 and Ans_2, respectively. The former subgoal call has already a completed subgoal frame, which is in the beginning of the chain, and a second subgoal frame being evaluated by thread 1 (thread 1 started the evaluation before the subgoal call have been completed by another thread). The latter subgoal call is still under evaluation by threads 1 and 2.

When comparing the previous design with the one in Fig. 4, one can easily observe that the new design has the following advantages: (i) requires less memory since it does not use a data structure based on trie nodes; (ii) at

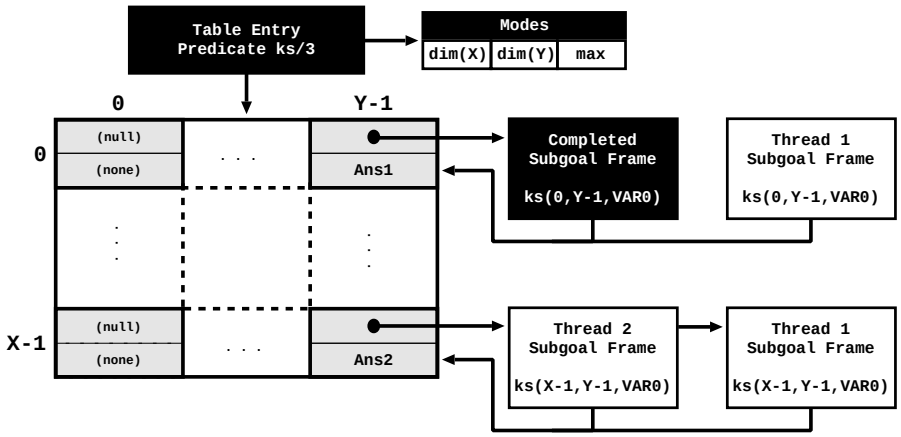

Fig. 4 The new multi-dimensional lock-free table space design

the subgoal representation level, it leads to less cache misses since, with a simple calculation, threads are able to access the bucket entry corresponding to the chain of subgoal frames, while in the previous design threads have to traverse at least one trie level to access such a chain; and (iii) at the answer representation level, a single field is enough to store the aggregated answer.

To support concurrency within the new table space design, we took advantage of the *CAS (Compare-And-Swap)* instruction, that nowadays can be widely found on many common architectures. The CAS operation is an *atomic instruction* that compares the contents of a memory location to a given value and, if they are the same, updates the contents of that memory location to a given new value. The atomicity guarantees that the new value is calculated based on up-to-date information, i.e., if the value had been updated by another thread in the meantime, the write would fail. Besides reducing the granularity of the synchronization, the CAS operation is at the heart of many *lock-free data structures* [7]. Lock-free data structures offer several advantages over their blocking counterparts, such as being immune to deadlocks, tolerant to priority inversion and convoying, kill-tolerant availability and preemption-tolerant [11]. As we will show next, our proposal was designed from scratch to be lock-free.

4 Algorithms

In this section, we discuss in more detail the key algorithms that implement the new table space design.

Algorithm 1 shows the pseudo-code for the process of obtaining the subgoal frame corresponding to a table entry TE and a subgoal call SC, given a thread identifier TID. The algorithm begins by getting the mode arguments (MA) and the bucket entry (BE) for the subgoal call SC (lines 1–2). Next, it tries to find a suitable subgoal frame sf (line 3), i.e., a completed subgoal frame or its own subgoal frame (allocated by a previous generator call to this procedure),

Algorithm 1 *CheckInsertSubgoalFrame(table entry TE, call SC, thread TID)*

```
 1: MA ← GetModeArguments(TE)
 2: BE ← GetBucketEntry(TE, SC, MA)
 3: sf ← FindCompletedOrThreadSubgoalFrame(BE, TID)
 4: if sf then {a completed subgoal frame or the thread's subgoal frame was found}
 5:    return sf
 6: else {no suitable subgoal frame was found}
 7:    nsf ← InitNewSubgoalFrame(TID)
 8:    repeat {get a completed subgoal frame or insert a new subgoal frame}
 9:       first_sg ← SubgoalFrameRef(BE)
10:       if IsCompleted(first_sg) then {completed subgoal frame found}
11:          ReleaseSubgoalFrame(nsf)
12:          return first_sg
13:       else
14:          NextRef(SF) ← first_sg
15:    until CAS(SubgoalFrameRef(BE), first_sg, nsf)
16:    return nsf
```

in which case the algorithm ends by returning it (lines 4–5). Otherwise, no suitable subgoal frame was found, thus a new temporary subgoal frame nsf is allocated and initialized with state *incomplete* for thread TID (line 7).

In the continuation, the algorithm tries to insert nsf in the chain of subgoal frames. To do that, it enters in a loop trying to insert the new subgoal frame in the beginning of the chain. Since, at the same time, another thread can be completing its own subgoal frame and moving it to the beginning of the chain, we need to guarantee synchronization between both operations. Therefore, the algorithm starts by obtaining the reference $first\_sg$ to the first subgoal frame in the chain (line 9) and rechecks if it refers a completed frame (that could have been completed in the meantime), in which case it releases the previously allocated frame and ends by returning the reference to the completed frame (lines 10–12). Otherwise, the algorithm tries to insert nsf in the chain of subgoal frames. For that, it updates its $NextRef()$ field to point to the current frame on the beginning of the chain (line 14) and then tries to insert nsf in the head of the chain by using a CAS operation (line 15). If the CAS operation succeeds, then nsf becomes a permanent frame and the algorithm ends by returning it (line 16). Otherwise, if the CAS fails, that means that another thread has updated the head of the chain in the meantime. In this case, the algorithm reads the new head reference $first\_sg$ and the process is restarted.

Upon completion of a particular subgoal call, which happens when all answers are computed, a thread calls Alg. 2 to update the corresponding subgoal frame SF to the *completed state*. Algorithm 2 begins by making a copy of SF to a new frame (nsf) and by marking SF to be removed from the chain of subgoal frames for the call at hand (lines 1–2). Next, nsf is marked as complete and its next chain reference is updated to $Null$ (lines 3–4). If successfully inserted in the chain for the call at hand, nsf will be the entry point of the chain to indicate that the subgoal call is completed. In the continuation, the algorithm gets the bucket entry BE from SF (line 5) and enters in a CAS loop (lines 6–11). The CAS loop will end in one of two situations: (i) another

Algorithm 2 *MoveToCompleted(subgoal frame SF)*

```
1: nsf ← CopySubgoalFrame(SF)
2: MarkForRemovalSubgoalFrame(SF)
3: MarkAsCompleted(nsf)
4: NextRef(nsf) ← Null
5: BE ← GetBucketEntry(SF)
6: repeat {trying to move the subgoal frame to the head of the chain}
7:     first_sg ← SubgoalFrameRef(BE)
8:     if IsCompleted(first_sg) then {a completed subgoal frame already exists}
9:         ReleaseSubgoalFrame(nsf)
10:        return
11: until CAS(EntryRef(BE), first_sg, nsf)
12: return
```

completed subgoal frame, inserted in the meantime by another thread, is found in the beginning of the chain (line 8–10); (ii) the subgoal frame nsf is successfully inserted in the head of the chain (line 11). In situation (i), nsf becomes useless and is thus released (line 9). In situation (ii), nsf becomes the single permanent subgoal frame since its next reference was previously set to $Null$.

Our implementation uses a copying technique to ensure that the completed frame is stored in the beginning of the chain since, otherwise, this constraint cannot be guaranteed. A problematic situation occurs when the subgoal frame SF being updated to completed is already the first on the chain and, concurrently, a second thread U executing Alg. 1 is trying to insert another frame SF_2 in the beginning of the chain. If we simply try to update SF to completed (without the copying technique), it might happen that SF_2 is inserted in the beginning of the chain just before SF be updated to completed, which violates the constraint of having the completed frame in the beginning of the chain. In more detail, thread U might have seen SF still as not completed (line 10 in Alg. 1) and then successfully insert SF_2 in the beginning of the chain using the CAS operation in line 15 of Alg. 1.

Next, we describe the algorithms used to generate and consume answers to/from a subgoal call. Algorithm 3 shows the pseudo-code for the process of updating the table space when a new answer ANS is found for a generator node N. The algorithm begins by obtaining the corresponding subgoal frame SF and bucket entry BE for the generator N (lines 1–2). Next, it checks if this is the first answer for SF and, if it is, it tries to insert ANS using a CAS operation and returns (lines 3–5). The first answer is always a correct answer for any aggregator mode. Otherwise, if it is not the first answer or if the CAS failed, then at least one answer already exists. Thus, the algorithm gets the mode aggregator for SF (line 6) and proceeds by computing the new answer according with the mode at hand. For simplicity of presentation, we only show the case of the *max* mode aggregator (lines 7–13). The other modes are treated similarly. For the *max* mode, the algorithm then tries to update BE with ANS if it is greater than the current answer in BE. To do so, it repeats a CAS operation until it succeeds (lines 8–12) or until it finds a better (maximal) answer, case where it simply returns such answer (lines 10–11).

Algorithm 3 *CheckInsertAnswer(answer ANS, generator N)*

```
 1: SF ← GetSubgoalFrame(N)
 2: BE ← GetBucketEntry(SF)
 3: if HasNoAnswer(BE) then {first answer}
 4:     if CAS(Answer(BE), Null, ANS) then {answer inserted}
 5:         return ANS
 6: aggregator ← GetAggregatorMode(SF)
 7: if aggregator = AGGREGATOR_MAX then
 8:     repeat {try to insert the answer if greater than the current one}
 9:         current_ans ← Answer(BE)
10:         if ANS ≤ current_ans then
11:             return current_ans
12:     until CAS(Answer(BE), current_ans, ANS)
13:     return ANS
14: else if aggregator = ... then {remaining aggregator modes}
15:     ...
```

Finally, Alg. 4 presents the pseudo-code for the process of loading an answer to a consumer node N. As for Alg. 3, it begins by obtaining the corresponding subgoal frame SF and bucket entry BE for the consumer node N (lines 1–2). Next, it checks if the last consumed answer in N is different from the one stored in the table space from the call at hand, i.e., if new answers were found since the last consumed answer marked in the field $LastConsumedAnswer()$ (line 3). If this is the case, then the $LastConsumedAnswer()$ field is updated accordingly and the new answer returned to the consumer node N (lines 4–5). Otherwise, if no new answers exist, it simply returns a $Null$ reference. It is important to note that the answer being returned is the one in the field $LastConsumedAnswer()$ of N and not the one in the $Answer()$ field of BE. This is because it might happen that the answer in BE could be updated, in the meantime, by another thread between the instant that it was read (line 4) and the instant that the return operation was executed (line 5), causing the current executing thread to miss the consumption of an answer.

Algorithm 4 *CheckConsumeAnswer(consumer N)*

```
1: SF ← GetSubgoalFrame(N)
2: BE ← GetBucketEntry(SF)
3: if LastConsumedAnswer(N) ≠ Answer(BE) then {new (unconsumed) answer}
4:     LastConsumedAnswer(N) ← Answer(BE)
5:     return LastConsumedAnswer(N)
6: else {no new answers}
7:     return Null
```

5 Correctness

In this section, we discuss the correctness of our proposal. Its full proof consists in two parts: first prove that the proposal is *linearizable* and then prove that

progress occurs in a lock-free fashion. Due to the lack of space, we focus on the linearization proof and we describe the linearization points of the proposal, the set of invariants and parts of the proof that the linearization points *preserve* the set of invariants. The linearization points in the algorithms shown are:

LP_1 *CheckInsertSubgoalFrame()* is linearizable at the CAS operation in line 15.
LP_2 *MoveToCompleted()* is linearizable at the *MarkForRemovalSubgoalFrame()* procedure in line 2.
LP_3 *MoveToCompleted()* is linearizable at the CAS operation in line 11.
LP_4 *CheckInsertAnswer()* is linearizable at the CAS operation in line 4.
LP_5 *CheckInsertAnswer()* is linearizable at the CAS operation in line 12.

The set of invariants that must be *preserved on every state* are:

Inv_1 A chain of subgoal frames always ends in a $Null$ reference.
Inv_2 The reference to the next in chain of a subgoal frame SF_1, corresponding to a subgoal call SC, must always refer to: (i) $Null$; (ii) another subgoal frame SF_2 also corresponding to SC.
Inv_3 The allocation of a subgoal frame SF must comply with the following semantics: (i) the initial state is temporary; (ii) the follower state is permanent if SF is inserted in a chain; (iii) the final state is released.
Inv_4 The visibility of a subgoal frame SF must comply with the following semantics: (i) the initial state is visible to a single thread; (ii) the follower state is visible to all threads if SF is inserted in a chain; (iii) the final state is invisible if SF does not belong to any chain.
Inv_5 The state of a subgoal frame SF must comply with the following semantics: (i) the initial state is incomplete; (ii) the final state is complete.
Inv_6 A subgoal call SC in evaluation has at least one subgoal frame SF in its chain.
Inv_7 A subgoal call SC fully evaluated has at least one completed subgoal frame SF in its chain.
Inv_8 A subgoal frame SF marked as completed and allocated as permanent is always in the beginning of the chain and visible to all threads.
Inv_9 Given a mode aggregator MA and a sequence S of answers for a subgoal call SC, the answer stored for SC is always the answer corresponding to the application of MA to S.
Inv_{10} Given a mode aggregator MA and a subgoal call SC, then an aggregated answer A_1 is only stored once for SC, i.e., once A_1 is replaced by another answer A_2, then A_1 is never again the aggregated answer for SC.
Inv_{11} Given a mode aggregator MA, a subgoal call SC and the sequence S of all answers for SC, then the final aggregated answer for SC is always consumed by all consumer nodes.

Finally, we show the proof on how two of the linearization points, namely LP_1 and LP_4, *preserve* the set of invariants. As mentioned before, due to the lack of space, we cannot show the proof for the remaining linearization points, but they follow a similar proof strategy.

Lemma 1 *Linearization point LP_1 preserves the set of invariants.*

Proof This linearization point refers to the insertion of a new subgoal frame nsf in the bucket entry of a subgoal call SC. Previous to the execution of the CAS in line 15, nsf is: (i) in a incomplete state (by Inv_5); and (ii) referring to $first\_sg$ (line 14) which is $Null$ or another subgoal frame (by Inv_1 and Inv_2). After the successful execution of the CAS operation, Inv_1, Inv_2 and Inv_5 hold, because the reference and the state of nsf remain unchanged. Inv_3, Inv_4 and Inv_6 also hold because with the insertion of nsf in the chain of subgoal frames for SC, nsf passes from a temporary state (only visible to thread TID) to a permanent state visible to all threads, meaning that at least one subgoal frame is in the chain of subgoal frames for SC.

Finally, we prove that Inv_8 also holds. To do so, we must ensure that $first\_sg$ is not complete up to the successful execution of the CAS operation. Assume that thread TID has just set $first\_sg$ in line 9 and is prepared to execute line 10. If $first\_sg$ is complete then TID would execute lines 11–12 and return (and would not have been able to reach the CAS operation). Otherwise, if $first\_sg$ is not complete in line 10 then it will never be the *completed subgoal frame in the beginning of the chain*. This is true because in Alg. 2 we specifically create a new complete subgoal frame and use it in linearization point LP_3, whenever the subgoal call SC is complete. Thus, up to the execution of the CAS operation either (i) SC has a complete subgoal frame in the beginning of the chain and since it is necessarily different from $first\_sg$, the CAS fails; or (ii) $first\_sg$ is still in the beginning of the chain and it is not complete. In both scenarios, Inv_8 holds. The remaining invariants are not affected. □

Lemma 2 *Linearization point LP_4 preserves the set of invariants.*

Proof This linearization point refers to the insertion of the first answer in a bucket entry BE. Previous to the execution of the CAS in line 4, BE does not have any answer. After the successful execution of the CAS operation, Inv_9 and Inv_{10} hold, because the answer ANS is a valid answer, since it was found during the evaluation of the subgoal call at hand, and any mode aggregator applied to a single answer ANS results in the insertion of ANS in the table. The remaining invariants are not affected. □

6 Performance Analysis

In this section, we present a performance analysis of our new multi-dimensional lock-free table space design. The environment of our experiments was a machine with 32-core AMD Opteron (tm) Processor 6274 @ 2.2 GHz with 32 GBytes of main memory and running the Linux kernel 3.18.6-100.fc20.x86_64 and YAP Prolog 6.3.2. We focused on two well-known dynamic programming problems, the Knapsack and the Longest Common Subsequence (LCS) problems. For the Knapsack problem, we fixed the number of items and capacity,

Table 1 Execution time, in milliseconds, for the sequential version ($\mathbf{T}_{seq}$) and the multi-threaded version with one thread ($\mathbf{T}_1$) and the corresponding ratio between the two versions ($\mathbf{T}_1/\mathbf{T}_{seq}$) for the top-down and bottom-up approaches of the Knapsack and LCS problems using YAP with the subgoal sharing design with shared answers of completed subgoals (**SS**) and using YAP with the new multi-dimensional lock-free table space design (**MD**)

Approach & Dataset		Subgoal Sharing (SS) Time $\mathbf{T}_{seq}$	Subgoal Sharing (SS) Time $\mathbf{T}_1$	Subgoal Sharing (SS) Ratio $\mathbf{T}_1/\mathbf{T}_{seq}$	Multi-Dimensional (MD) Time $\mathbf{T}_{seq}$	Multi-Dimensional (MD) Time $\mathbf{T}_1$	Multi-Dimensional (MD) Ratio $\mathbf{T}_1/\mathbf{T}_{seq}$	SS vs MD Ratio $\mathbf{SS}_{T_1}/\mathbf{MD}_{T_1}$
Knapsack Problem								
$\mathbf{TD}_{no}$	$\mathbf{D}_{10}$	9,508	12,415	1.31	4,275	5,241	1.23	2.37
	$\mathbf{D}_{30}$	9,246	12,177	1.32	4,196	5,336	1.27	2.28
	$\mathbf{D}_{50}$	9,480	12,589	1.33	4,275	5,457	1.28	2.31
$\mathbf{TD}_{rnd}$	$\mathbf{D}_{10}$	19,667	24,444	1.24	10,462	11,740	1.12	2.08
	$\mathbf{D}_{30}$	19,847	25,609	1.29	10,508	11,959	1.14	2.14
	$\mathbf{D}_{50}$	19,985	25,429	1.27	10,805	11,982	1.11	2.12
BU	$\mathbf{D}_{10}$	12,614	17,940	1.42	7,001	7,668	1.10	2.34
	$\mathbf{D}_{30}$	12,364	17,856	1.44	7,005	7,786	1.11	2.29
	$\mathbf{D}_{50}$	12,653	17,499	1.38	6,775	7,637	1.13	2.29
LCS Problem								
$\mathbf{TD}_{no}$	$\mathbf{D}_{10}$	21,191	26,225	1.24	16,202	18,046	1.11	1.45
	$\mathbf{D}_{30}$	20,809	26,146	1.26	16,006	18,067	1.13	1.45
	$\mathbf{D}_{50}$	20,775	26,028	1.25	16,259	18,195	1.12	1.43
$\mathbf{TD}_{rnd}$	$\mathbf{D}_{10}$	34,565	44,371	1.28	21,525	23,635	1.10	1.88
	$\mathbf{D}_{30}$	34,284	44,191	1.29	21,512	24,055	1.12	1.84
	$\mathbf{D}_{50}$	33,989	44,158	1.30	21,477	23,736	1.11	1.86
BU	$\mathbf{D}_{10}$	20,799	28,909	1.39	11,453	14,017	1.22	2.06
	$\mathbf{D}_{30}$	21,174	28,904	1.37	11,218	14,189	1.26	2.04
	$\mathbf{D}_{50}$	21,166	28,857	1.36	11,139	13,982	1.26	2.06

respectively, 1,600 and 3,200. For the LCS problem, we used sequences with a fixed size of 3,200 symbols. Then, for each problem, we implemented either *top-down* (**TD**) and *bottom-up* (**BU**) approaches. For the top-down approaches, we tested both problems without randomization ($\mathbf{TD}_{no}$) and with randomization using Stivala et al.'s approach [18] with an extra random displacement clause ($\mathbf{TD}_{rnd}$). We also created three different datasets for each approach, $\mathbf{D}_{10}$, $\mathbf{D}_{30}$ and $\mathbf{D}_{50}$, meaning that the values for the weights/profits for the Knapsack problem and the symbols for the LCS problem were randomly generated in an interval between 1 and 10%, 30% and 50% of the total number of items/symbols, respectively[3].

Table 1 shows the execution time for the sequential ($\mathbf{T}_{seq}$) and multi-threaded version with one thread ($\mathbf{T}_1$) for the several configurations of the Knapsack and LCS problems using YAP with the subgoal sharing design (**SS** in what follows) and with the new multi-dimensional lock-free design (**MD** in what follows). All execution times are the average of 10 runs. For $\mathbf{T}_{seq}$, YAP was compiled without multithreaded support and ran without multithreaded code. Columns $\mathbf{T}_1/\mathbf{T}_{seq}$ show the overhead of the multithreaded version over the sequential version and column $\mathbf{SS}_{T_1}/\mathbf{MD}_{T_1}$ compares the execution times for the multithreaded versions of both designs.

By analyzing the results on Table 1, we can observe that the new **MD** design clearly outperforms the previous **SS** design either for the sequential

[3] Datasets available at `https://github.com/miar/yap-6.3/tree/master/parallel_dynamic_programming`.

Table 2 Execution time, in milliseconds, for one thread ($\mathbf{T}_1$) and corresponding speedups ($\mathbf{T}_1/\mathbf{T}_p$) for the execution with 8, 16, 24 and 32 threads, for the top-down and bottom-up approaches of the Knapsack problem using YAP with the subgoal sharing design with shared answers of completed subgoals (**SS**) and using YAP with the new multi-dimensional lock-free table space design (**MD**)

Approach & Dataset		**Time ($\mathbf{T}_1$)**	**# Threads (p) Speedup ($\mathbf{T}_1/\mathbf{T}_p$)**				**Best Time ($\mathbf{T}_{best}$)**
		1	**8**	**16**	**24**	**32**	
Subgoal Sharing (SS)							
TD$_{no}$	**D**$_{10}$	**12,415**	n.c.	n.c.	n.c.	n.c.	12,415
	D$_{30}$	**12,177**	n.c.	n.c.	n.c.	n.c.	12,177
	D$_{50}$	**12,589**	n.c.	n.c.	n.c.	n.c.	12,589
TD$_{rnd}$	**D**$_{10}$	24,444	6.78	12.35	15.44	**18.19**	1,344
	D$_{30}$	25,609	7.15	13.83	17.37	**20.47**	1,251
	D$_{50}$	25,429	7.27	13.70	17.35	**20.62**	1,233
BU	**D**$_{10}$	17,940	7.17	13.97	18.31	**22.15**	0,810
	D$_{30}$	17,856	7.23	13.78	18.26	**21.94**	0,814
	D$_{50}$	17,499	7.25	14.01	18.34	**21.76**	0,804
Multi-Dimensional (MD)							
TD$_{no}$	**D**$_{10}$	**5,241**	n.c.	n.c.	n.c.	n.c.	5,241
	D$_{30}$	**5,336**	n.c.	n.c.	n.c.	n.c.	5,336
	D$_{50}$	**5,457**	n.c.	n.c.	n.c.	n.c.	5,457
TD$_{rnd}$	**D**$_{10}$	11,740	6.90	12.90	16.22	**19.09**	0,615
	D$_{30}$	11,959	7.31	14.04	18.01	**21.59**	0,554
	D$_{50}$	11,982	7.36	14.03	17.96	**21.63**	0,554
BU	**D**$_{10}$	7,668	7.24	13.77	17.55	**21.42**	0,358
	D$_{30}$	7,786	7.40	14.13	18.02	**22.18**	0,351
	D$_{50}$	7,637	7.37	13.96	18.10	**21.95**	0,348

execution and the multithreaded execution with one thread. On average, the **MD** design is around 2 to 2.5 times faster for the Knapsack problem and around 1.5 to 2.0 times faster for the LCS problem than the **SS** design (results on column $\mathbf{SS}_{T_1}/\mathbf{MD}_{T_1}$). Additionally, the overheads introduced by the multithreaded version over the sequential version also seem to be less relevant in the **MD** design than in the **SS** design. On average, the overheads introduced by the **MD** design are around 10% to 20%, while the overheads for the **SS** design are around 30% to 40% (results on columns $\mathbf{T}_1/\mathbf{T}_{seq}$). In summary, the results in Table 1 show that the new **MD** design introduces less overheads than the **SS** design and clearly improves the execution time for sequential and multithreaded execution with one thread.

Table 2 and Table 3 then show results for the execution with multiple threads for the top-down and bottom-up approaches, respectively for the Knapsack and LCS problems. Column $\mathbf{T}_1$ repeats the results for the execution time with one thread and columns $\mathbf{T}_1/\mathbf{T}_p$ show the corresponding speedup for the execution with 8, 16, 24 and 32 threads. For each configuration, the results in bold highlight the column where the best execution time was obtained and the last column ($\mathbf{T}_{best}$) presents such result in milliseconds.

Analyzing the general picture in both tables, one can observe that the $\mathbf{TD}_{rnd}$ top-down and **BU** bottom-up approaches have the best results with excellent speedups for 8, 16, 24 and 32 threads. In particular, with 32 threads, they obtain speedups between 18 and 22, for both problems and designs.

Table 3 Execution time, in milliseconds, for one thread ($\mathbf{T}_1$) and corresponding speedups ($\mathbf{T}_1/\mathbf{T}_p$) for the execution with 8, 16, 24 and 32 threads, for the top-down and bottom-up approaches of the LCS problem using YAP with the subgoal sharing design with shared answers of completed subgoals (**SS**) and using YAP with the new multi-dimensional lock-free table space design (**MD**)

Approach & Dataset		**Time ($\mathbf{T}_1$)**	**# Threads (p) Speedup ($\mathbf{T}_1/\mathbf{T}_p$)**				**Best Time ($\mathbf{T}_{best}$)**
		1	**8**	**16**	**24**	**32**	
Subgoal Sharing (SS)							
TD$_{no}$	**D**$_{10}$	**26,225**	n.c.	n.c.	n.c.	n.c.	26,225
	D$_{30}$	**26,146**	n.c.	n.c.	n.c.	n.c.	26,146
	D$_{50}$	**26,028**	n.c.	n.c.	n.c.	n.c.	26,028
TD$_{rnd}$	**D**$_{10}$	44,371	7.23	13.23	16.45	**19.74**	2,248
	D$_{30}$	44,191	7.12	13.09	16.52	**19.77**	2,235
	D$_{50}$	44,158	7.06	13.30	16.49	**19.58**	2,255
BU	**D**$_{10}$	28,909	6.47	12.21	16.48	**20.32**	1,423
	D$_{30}$	28,904	6.94	12.61	16.63	**20.40**	1,417
	D$_{50}$	28,857	6.44	12.31	16.44	**20.52**	1,406
Multi-Dimensional (MD)							
TD$_{no}$	**D**$_{10}$	**18,046**	n.c.	n.c.	n.c.	n.c.	18,046
	D$_{30}$	**18,067**	n.c.	n.c.	n.c.	n.c.	18,067
	D$_{50}$	**18,195**	n.c.	n.c.	n.c.	n.c.	18,195
TD$_{rnd}$	**D**$_{10}$	23,635	7.31	13.60	17.57	**21.25**	1,112
	D$_{30}$	24,055	7.46	14.03	17.99	**21.40**	1,124
	D$_{50}$	23,736	7.33	13.76	17.78	**21.23**	1,118
BU	**D**$_{10}$	14,017	6.90	12.14	16.89	**22.21**	0,631
	D$_{30}$	14,189	6.88	13.10	17.01	**22.49**	0,631
	D$_{50}$	13,982	6.87	13.06	16.85	**22.55**	0,620

Columns $\mathbf{T}_{best}$ also show that, for a particular problem and design, the **BU** approach running with 32 threads obtains the best execution times of all configurations. The speedup results for the **TD**$_{no}$ approach were *not considered* (*n.c.*) since without randomization this approach is unable to take advantage of our framework (all threads would replicate the same evaluation sequence and, thus, they would not be able to reuse the answers from the other threads).

Comparing now the results for both designs, we can observe that, in general, the **MD** design keeps the same speedups ratios despite its base execution times (with one thread) being 1.5 to 2.5 times faster than the **SS** design, as the results on Table 1 show. In particular, for the **TD**$_{rnd}$ and **BU** approaches, the speedup results are slightly better in favor of the **MD** design. For the execution times, the **MD** design is clearly better by far. For example, consider the Knapsack problem with the **BU** approach and the **D**$_{50}$ dataset with 32 threads, one can observe that the **MD** design takes 0,348 milliseconds, while the **SS** design requires 0,804 milliseconds, i.e., more than a half reduction in the absolute execution time. The same situation occurs in the LCS problem with the **BU** approach and the **D**$_{50}$ dataset with 32 threads, where the **MD** design runs in 620 milliseconds, while the **SS** design runs in 1,406 milliseconds.

To understand better these results, we have also collected internal statistics regarding both designs. According to those statistics, the better performance results of the **MD** design are mainly due to three reasons. The first reason is because the **MD** design implements lock-freedom on the complete table space,

while the **SS** design implements lock-freedom only within the subgoal tries. Lock-free techniques are known to achieve very good performances in highly concurrent environments. In our new design, this means that threads are able to access subgoal frames and read answers without any locking mechanism, while insertions are all done using the CAS low-level instruction. The second reason is because the answers found for subgoal calls are immediately shared by all threads during the evaluation, while in the **SS** design, the answers are only shared after the subgoal call is complete. The third and last reason is the very efficient and compact representation of the table space. To support this claim, we have collected the memory footprint of the **SS** and **MD** designs, on both the Knapsack and LCS problems, when using the $\mathbf{D}_{50}$ dataset with the top-down and bottom-up parallelization strategies. The memory footprint was collected after the complete evaluation of all subgoal calls, so that the complete table space size could be the same for any number of threads launched. For the **SS** design, the memory used on the subgoal and answer tries was 704, 695 and 695 MBytes on the Knapsack problem, and 1407, 1406 and 1407 MBytes on the LCS problem, for the $\mathbf{TD}_{no}$, $\mathbf{TD}_{rnd}$ and **BU** approaches, respectively. To store the same tables, the **MD** design used about 9 times less memory on all approaches. Consequently, in the **MD** design, threads access less data structures and memory positions, while in the **SS** design, threads have to traverse the trie data structures for both tabled calls and answers, leading also to a high ratio of penalties due to potential cache misses and page faults.

7 Conclusions and Further Work

The key contribution of this work is a new design at the underlying tabling engine specially aimed to support the efficient handling of multi-dimensional arrays. We propose a new mode for indexing arguments in mode-directed tabled evaluations, named *dim*, where each *dim* argument features a uni-dimensional lock-free concurrent array. This allows users to explicitly define which tabled predicates could benefit from a more compact design when aggregating solutions for multiple integer dimensions.

To show the potential of the new design, we used two well-known dynamic programming problems and we discussed how we were able to reduce their execution times and scale the execution, using either top-down and bottom-up techniques. Experimental results, on a 32-core AMD machine, showed that the new design introduces less overheads than the previous design and clearly improves the execution time for sequential and multithreaded execution. In particular, for multithreaded execution up to 32 threads, the new design showed to be able to maintain or achieve slightly better speedups despite its base execution times (with one thread) be 1.5 to 2.5 times faster than the previous design. As further work, we pretend to apply the new design to other dynamic programming problems and explore its impact in other application domains.

References

1. Areias, M., Rocha, R.: Towards Multi-Threaded Local Tabling Using a Common Table Space. Journal of Theory and Practice of Logic Programming **12**(4 & 5), 427–443 (2012)
2. Areias, M., Rocha, R.: On Scaling Dynamic Programming Problems with a Multi-threaded Tabling System. Journal of Systems and Software **125**, 417–426 (2017)
3. Bellman, R.: Dynamic Programming. Princeton University Press (1957)
4. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1), 20–74 (1996)
5. Dawson, S., Ramakrishnan, C.R., Warren, D.S.: Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In: ACM Conference on Programming Language Design and Implementation, pp. 117–126. ACM (1996)
6. Guo, H.F., Gupta, G.: Simplifying Dynamic Programming via Mode-directed Tabling. Software Practice and Experience **38**(1), 75–94 (2008)
7. Herlihy, M., Wing, J.M.: Axioms for Concurrent Objects. In: ACM Symposium on Principles of Programming Languages, pp. 13–26. ACM (1987)
8. Johnson, M.: Memoization in Top-Down Parsing. Computational Linguistics **21**(3), 405–417 (1995)
9. Louka, B., Tchuente, M.: Dynamic Programming on Two-dimensional Systolic Arrays. Information Processing Letters **29**(2), 97–104 (1988)
10. Marques, R., Swift, T.: Concurrent and Local Evaluation of Normal Programs. In: International Conference on Logic Programming, no. 5366 in LNCS, pp. 206–222. Springer (2008)
11. Michael, M.M.: Scalable Lock-free Dynamic Memory Allocation. ACM SIGPLAN Notices **39**(6), 35–46 (2004)
12. Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Swift, T., Warren, D.S.: Efficient Model Checking Using Tabled Resolution. In: Computer Aided Verification, no. 1254 in LNCS, pp. 143–154. Springer (1997)
13. 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), 31–54 (1999)
14. Rocha, R., Fonseca, N.A., Santos Costa, V.: On Applying Tabling to Inductive Logic Programming. In: European Conference on Machine Learning, no. 3720 in LNAI, pp. 707–714. Springer (2005)
15. Rytter, W.: On Efficient Parallel Computations for Some Dynamic Programming Problems. Theoretical Computer Science **59**(3), 297– 307 (1988)
16. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: ACM International Conference on the Management of Data, pp. 442–453. ACM (1994)
17. Santos, J., Rocha, R.: On the Efficient Implementation of Mode-Directed Tabling. In: International Symposium on Practical Aspects of Declarative Languages, no. 7752 in LNCS, pp. 141–156. Springer (2013)
18. Stivala, A., Stuckey, P., de la Banda, M.G., Hermenegildo, M., Wirth, A.: Lock-Free Parallel Dynamic Programming. Journal of Parallel and Distributed Computing **70**(8), 839–848 (2010)
19. Swift, T., Warren, D.S.: Tabling with Answer Subsumption: Implementation, Applications and Performance. In: European Conference on Logics in Artificial Intelligence, no. 6341 in LNAI, pp. 300–312. Springer (2010)
20. Wielemaker, J.: Native Preemptive Threads in SWI-Prolog. In: International Conference on Logic Programming, no. 2916 in LNCS, pp. 331–345. Springer (2003)
21. Yang, G., Kifer, M.: Flora: Implementing an Efficient Dood System using a Tabling Logic Engine. In: Computational Logic, no. 1861 in LNCS, pp. 1078–1093. Springer (2000)
22. 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, vol. 2, pp. 213–218. IEEE Computer Society (2010)
23. Zou, Y., Finin, T.W., Chen, H.: F-OWL: An Inference Engine for Semantic Web. In: International Workshop on Formal Approaches to Agent-Based Systems, *LNCS*, vol. 3228, pp. 238–248. Springer (2004)