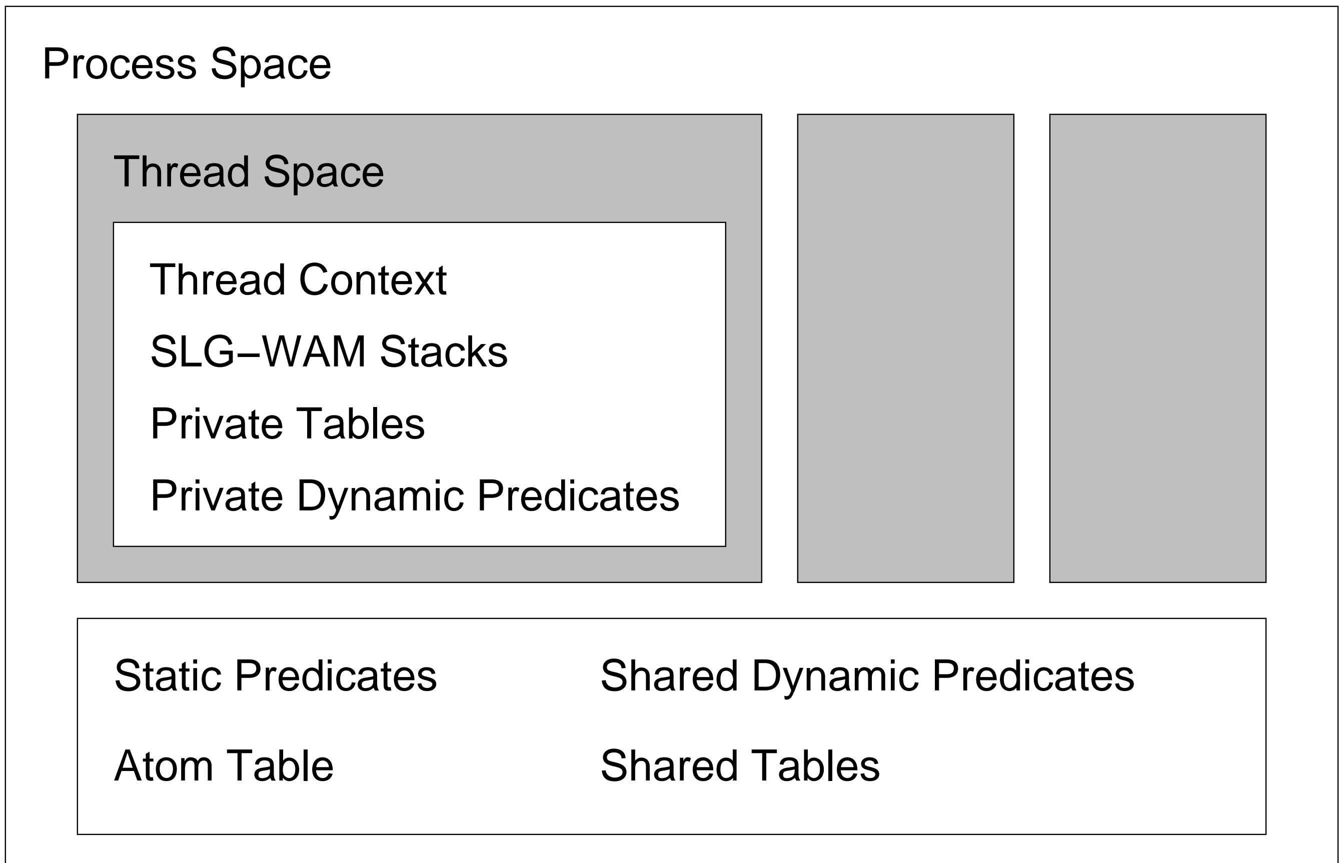
Concurrent Tabling for Definite Programs in XSB

Rui Marques

May 2nd, 2007

Multi-threaded XSB Process

- XSB process contains multiple threads
- Some data structures are thread private
- Others are shared among threads



Motivation for shared tables

- Memory efficiency
  - When N threads compute the same tables, using private tables will require N times more memory for table space the a single thread
- Amortization of computation time
  - The tables only needed the be computed once, even if they are used by different threads
- Parallel computation of tables
  - A program involving multiple tables may be computed by parallel threads
  - Each thread will compute a set of tables
  - *Table Parallelism* - a parallel logic programming paradigm,
    where the source of parallelism is the tabled subgoal

Shared Completed Tables
A first approach for sharing tables

- Tables are computed by only one thread
- When a thread tries to read from a table being computed by another thread, it suspends until the table is completed
- Will only allow parallel computation of subgoals that are do not depend on each other
  - If two subgoals that depend on each other are computed by different threads, one has to be completed before the second may proceed.
  - Consider the program:

```
:- table p/1, q/1.
p(1).               q(1).
p(2) :- q(X).       q(X) :- r(X).
p(3).
```

  - If thread 1 is computing `p(X)` and thread 2 is computing `q(X)`, thread 1 will suspend evaluating the second clause until thread 2 completes `q(X)`.

Shared Completed Tables Problem

- Shared completed table may produce deadlocks
- Consider the following program:

```
:- table a/1, b/1.
a(1).              b(1).
a(2).              b(3).
a(X) :- b(X).      b(X) :- a(X).
```

- Suppose that thread 1 is computing `a(X)` and thread 2 is computing `b(X)`.
- Thread 1 will suspend waiting for thread 2 to complete the table for `b(X)` and thread 2 will suspend waiting for thread 1 to complete `a(X)`.
- Deadlock!

Shared Completed Tables
Detecting deadlocks

- Define TDG (*Thread Dependency Graph*) as the graph where nodes are threads, and there is an edge when a thread is suspended waiting for a subgoal that belongs to another thread.
- TDG is a projection of the SDG (*the Sugbgoal Dependency Graph*).
- A deadlock occurs when there is a cycle in the TDG.
- Very easy to detect, because each thread may only wait for one subgoal to complete at a given moment.
- Detection is done right before suspending the thread.

Shared Completed Tables
Breaking deadlocks

- Once a deadlock is detected, the thread that detects the deadlock resolves the deadlock by grabbing all the subgoals that are known to belong to the SCC.
- This means that all the other threads will be reset to the point where they called the first goal on this SCC, in suspended state.
- This is only possible under local scheduling!
- The leader thread marks all the grabbed subgoals as belonging to itself and proceeds to compute the whole SCC.
- May lead to recomputation of subgoals!
- In the worst case all the subgoals an SCC with N subgoals computed by T threads may be recomputed T times.

Shared Completed Tables Implementation — Data Structures

- For each thread the following items are kept:
  - $WaitingForSubgoal$ — the subgoal the thread is waiting to complete
  - $WaitingForThread$ — thread whose the subgoal belongs — this is the edge on the TDG
  - $IsLeader$ — boolean field that is true if the thread is leader and will proceed computing the SCC.
  - $WasReset$ — boolean field use to signal the thread that it was reset by the leader
- For the subgoal frame the following items are added:
  - $OwnerThread$ — identifier of the thread whose this subgoal belongs to.
  - $Grabbed$ — boolean field that is true if the subgoal frame has been grabbed by the leader thread but is not yed being computed

TableTrySingle Instruction Revision

```
Instruction tabletrysingle(Arity, Subgoal_Trie_Root)      /* Subgoal is in argument registers */
  If (subgoal_check_insert(Subgoal,Subgoal_Trie_Root) == new)
(α)    /* Subgoal is new and added */
       Create and set up a subgoal frame SF for the Subgoal;
       Set up a generator choice point GCP to perform program clause resolution;
       Set the failure continuation FailCont cell of GCP to point to
                                                   a check_complete instruction;
       Push a new completion stack frame ComplSF onto the Completion Stack;
                                                   /* discussed below */
       Associate ComplSF with SF;
       Branch to the next instruction to perform program clause resolution;
  else  /* Subgoal was not new — already existed in the Subgoal_Trie */
       If (SF_ComplSF(Subgoal) == complete)
(β)    /* The subgoal frame has been marked as complete */
          Answer_Root := SF_AnsTrieRoot(Subgoal);
          Branch to Answer_Root to perform answer clause resolution
             by executing the code in the answer trie;
       else
(γ)    /* Subgoal is not new, and not complete */
          Create a consumer choice point CCP for Subgoal,
          and add CCP to the head of Subgoal's consumer choice point chain;
          Set the failure continuation FailCont cell of CCP to point to
                                                   an answer_return instruction;
          Call update_dependencies(Subgoal);     /* for scheduling ASCCs – discussed below */
          Freeze stacks and fail into CCP to execute answer_return instructions;
```

TableTrySingle Instruction Shared Completed Tables

```
Instruction tabletrysingle(Arity, Subgoal_Trie_Root)      /* Subgoal is in argument registers *
  If (subgoal_check_insert(Subgoal,Subgoal_Trie_Root) == new
        or SF_Grabbed(Subgoal) ) (α)            /* Subgoal is new and added */
     Create and set up a subgoal frame SF for the Subgoal; /* if not grabbbed */
     Set up a generator choice point GCP to perform program clause resolution;
     ...
  else  /* Subgoal was not new — already existed in the Subgoal_Trie */
     While (SF_ComplSF(Subgoal) != complete and
              SF_OwnerThread(Subgoal) != ThisThread )
        If DeadlockDetected()
           /* reset other threads stacks to the point where they first called this SCC */
           /* mark all subgoals there (including the current one)
            as grabbed and belonging to this thread */
        Else
           Suspend Thread()
        If ThreadWasReset()
           /* pcreg has the address of the instruction that must be re-executed
            so jump to next instruction */
     If (SF_ComplSF(Subgoal) == complete)
(β)     ...
     else
(γ)     ...
```

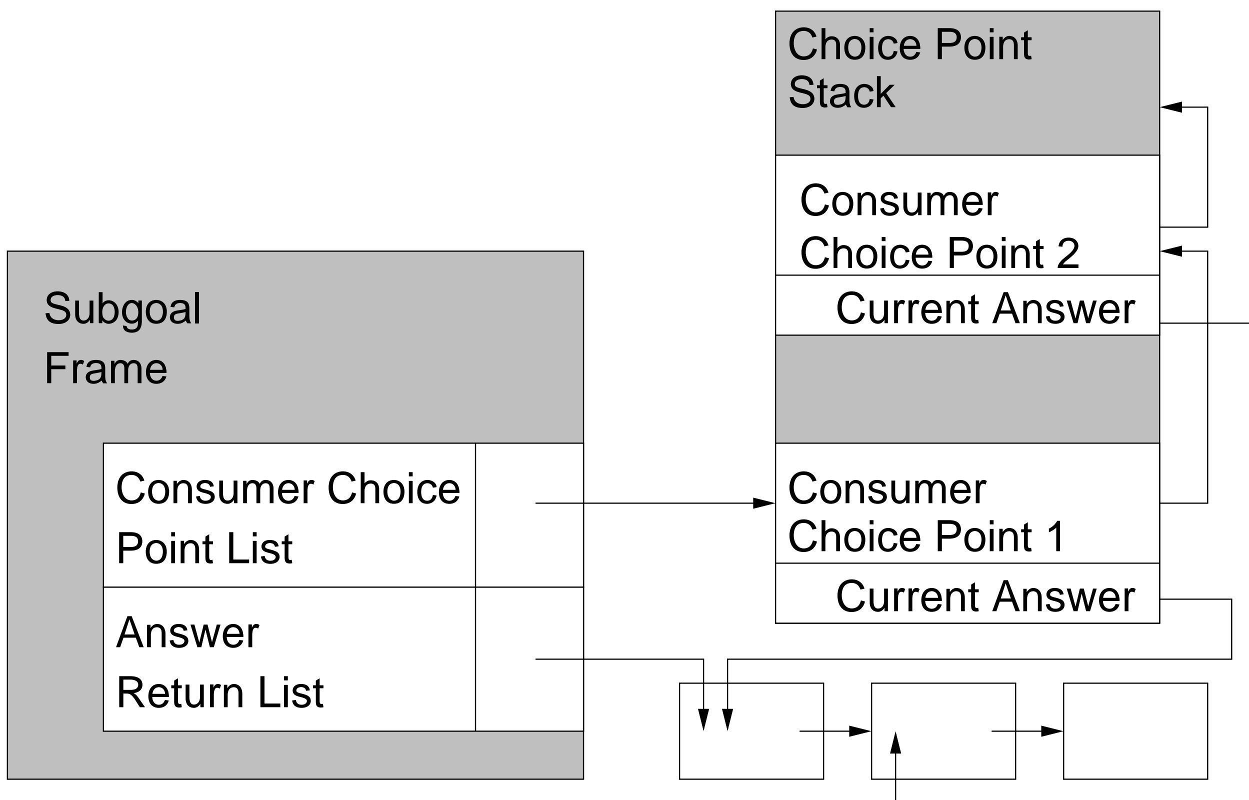
- The Completion instruction wakes up all the suspended threads!

Concurrent Completion Overview

- Allows threads to read tables belonging to other threads, while they are being computed.
- As such it supports table parallelism, where each thread computes its tables, and threads may co-operate to evaluate a program with multiple tables.
- Main burden is on the completion operation must detect that all threads computing an SCC are quiescent.
- While Shared Completed Tables require Local Scheduling, Concurrent Completion may be implemented over either Batched or Local.

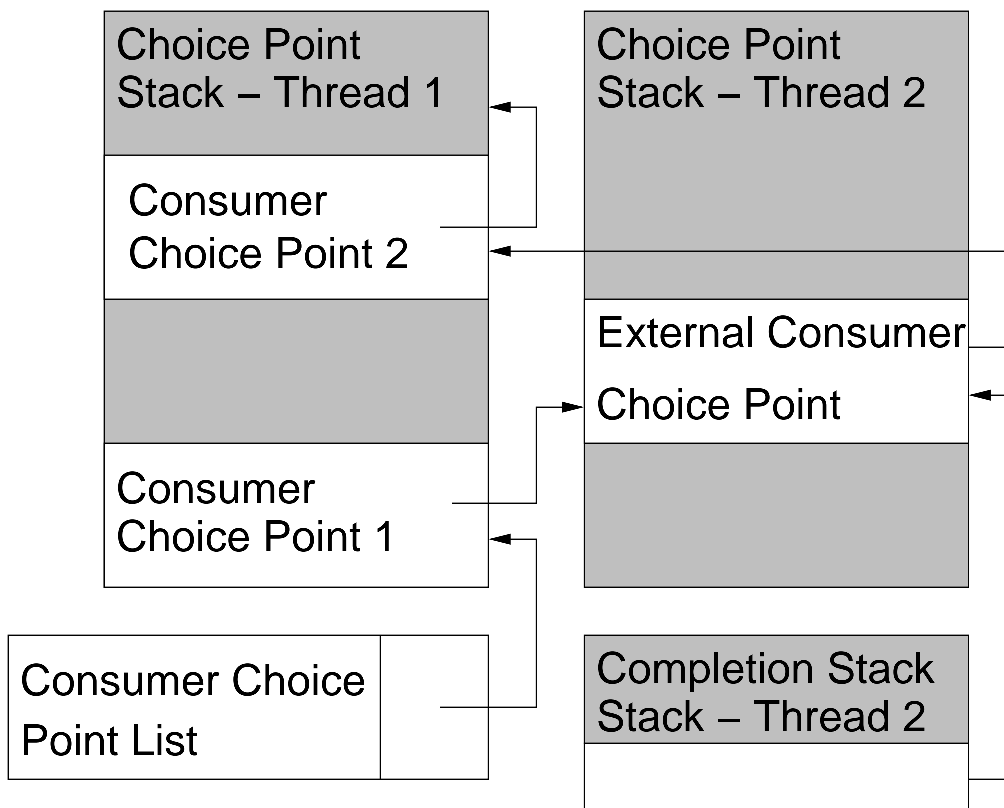
Tabling Data Structures
The Subgoal Frame

- For each subgoal there is a subgoal frame, which can be accessed through the *call trie*.
- The subgoal frame contains, among other stuff:
  - a pointer to the *answer return list* which contains the answers computed so far, also accessible through the *answer trie*.
  - a pointer to the *consumer choice point list* which contains the choice points that consume answers from this subgoal.



Tabling Data Structures
Concurrent Completion

- Reader threads choice point stacks contain *external consumer choice points* that are used to return answers among threads.
- The answer return list may only be written by the thread that owns the subgoal but may be read by reader threads.
- There is a frame on the completion stack for each external consumer choice point.



Concurrent Completion SLG-WAM Instructions

- `TableTry` as to cater with the new fields to the tabling data structures.
- `NewAnswer` and `AnswerReturn` have to support concurrency when accessing the answer return list.
- `CheckComplete` algorithm must be changed to detect quiescence of multi-threaded SCCs
  - Must be executed under mutual exclusion.
- At some point, threads must be notified that new answers were returned to them.
  - When completing at the `CheckComplete` instruction.
  - When scheduling at the `CheckComplete` instruction.
  - When returning a new answer at the `NewAnswer` instruction.

CheckComplete Instruction Revision

Instruction check_complete

0 $SubgComplStFr$:= SF_ComplSF(GCP_SubgFr($\mathbf{B}$));
/* Access Completion Stack via Subgoal Frame via GCP*/
1 If ($Subgoal$ is the leader of a scheduling ASCC, A)
1.1 Call fixpoint_check($SubgCSF$);
1.2.1 Mark as *complete* all subgoals in A;
1.2.2 Reclaim the stack space of subgoals in A and adjust the freeze registe
2 $\mathbf{B}$:= GCP_BregChain($\mathbf{B}$);
3 fail;

Procedure fixpoint_check($SubgCSF$) /* $SubgCSF$ is a pointer to the completion
while ($SubgCSF$ is less than or equal to the top of the completion stack)
$SubgFr$:= CSF_SubgFr($SubgCSF$);
Call schedule_resumes($SubgFr$);/* a failure continuation is taken if any cons
/* point associated with $SubgFr$ has unconsumed answers */
Increment $SubgCSF$ by the size of a completion stack frame;

Concurrent Completion
New data items for each thread

- *HasCompleted* — boolean field that signals that the SCC that the thread was waiting for has been completed.
- *IsCompleting* — boolean field that signals that the thread is awaiting completion.
- *Round* — counter that is incremented every time new answers are found.
- *TDL* — the *Thread Dependency List*. Contains an entry for each subgoal that the thread depends on. If it depends on more than one subgoal that belong to the same thread.
  It also keeps track of the *Round* field for every other thread, at the point of the last suspension.

Concurrent Completion
New Completion Instruction

Instruction check_complete

```
0    SubgComplStFr := SF_ComplSF(GCP_SubgFr(B));
While True
      If (Subgoal is not the leader of a scheduling ASCC, A)
           Break While;
      Call fixpoint_check(SubgCSF);
      If FixPoint not found
           Break While ;
      Extend Dependency List to cover all indirect dependencies ;
      If a Back Dependency has been found
           Update the ASCC information in the completion stack
           effectively changing the leader for this thread ;
           Break While ;
      If all threads in TDL are completing
           Check Round fields on TDL to see if any thread
           may have answers to return ;
           If not and TDL is a proper SCC
                Mark all subgoals as completed on the top ASCC of every
                thread in the TDL
                Signal all threads in TDL as completed and wake them up
                Break While ;
           Else wake all threads in the TDL
      Suspend thread ;
      If thread was completed
           Break While ;
B := GCP_BregChain(B);
fail;
```

Concurrent Completion
Changes to `fixpoint_check`

```
Procedure fixpoint_check(SubgCSF)  /* SubgCSF is a pointer to the completion
  while (SubgCSF is less than or equal to the top of the completion stack)
     SubgFr := CSF_SubgFr(SubgCSF);
     Call schedule_resumes(SubgFr);/* a failure continuation is taken if any cons
                        /* point associated with SubgFr has unconsumed answers */
     If SubgCSF corresponds to an external choice point frame
                        and has answers to return
                        add dependency into SubgFr.ThreadOwner to TDL
     Increment SubgCSF by the size of a completion stack frame;
```

Experimental Results

- Overhead
  - Varies widely among different systems
  - In some (e.g. Fedora Linux) its quite bad (in some benchmarks its over 50%).
  - We believe the slowdown is due to the fine grained locking mechanism used.
- Amortization
  - Re-using shared tables allows almost constant time complexity as opposed to linear complexity in the number of threads, using private tables. Experimental results confirm this.
- Parallelism
  - Experimental results didn't show improvements in the speed of execution of shared tables in multi-processors up to eight processors.
    This is true for both, Shared Completed Tables and Concurrent Completion.