On the Implementation of Speculative Constraint Processing

Jiefei Ma[1], Alessandra Russo[1], Krysia Broda[1]
Hiroshi Hosobe[2], and Ken Satoh[2]

[1] Imperial College London, United Kingdom
{jm103,ar3,kb}@doc.ic.ac.uk
[2] National Institute of Informatics, Japan
{hosobe,ksatoh}@nii.ac.jp

Abstract. Speculative computation has been proposed for reasoning with incomplete information in multi-agent systems. This paper presents the first multi-threaded implementation for speculative constraint processing with iterative revision for disjunctive answers in master-slave multi-agent systems.

1 Introduction

In the context of distributed problem solving with multi-agent systems, communication among agents plays a very important role, as it enables coordination and cooperation between agents. However, in practice communication is not always guaranteed. For example, the physical channel may delay/lose messages, or agents may break down or take unexpectedly long time to compute answers. Moreover, agents are often unable to distinguish between the above situations. All such problems/uncertainties can seriously affect the system performance, especially for result-sharing applications. For example, in a multi-agent scheduling problem, if some agents cannot respond to the queries of their local resources in time, then the computation of the overall resource assignment will be impossible or delayed.

Speculative computation has been proposed in [1,2,3,4,5] as a solution to the problem. In the proposal, a *master* agent prepares default answers to the questions that it can ask to the *slaves*. When communication is delayed or failed, the master can use the default answers to continue the computation. If later a real answer is returned (e.g. the communication channel or the slave agent is recovered), the computation already done by the master, which is using the default answers, will be revised. One of the main advantages of speculative computation relies then on the fact that the computation process of an agent is never halted when waiting for other agent's responses. Examples of real life situations where speculative computation is useful can be found in [1,2,3,4,5].

Within the last few years, speculative computation has gone through various stages of development and extensions. In [1] an abductive-based algorithm has been proposed for speculative computation with yes/no answers for master-slave systems. In [2], the algorithm has been generalised for hierarchical multi-agent systems where agents are assumed to be organised into a hierarchy of

J. Dix, M. Fisher, and P. Novák (Eds.): CLIMA X, LNAI 6214, pp. 178–195, 2010.

master/slaves. The method proposed in [2] also considers only yes/no type of answers. This approach has been extended in [3] to allow more general queries, whereby an agent can ask *possible values* or *constraints* of given queries, but within the context of master-slave systems. This speculative constraint processing takes into account the possibility that the agent's response may neither entail nor contradict the default answer assumed during the computation. In this case the two alternative computations – the one that uses the default and the one that uses the agent's response – are maintained active. The approach described in [3] assumes, however, that only the master agent can perform speculative computation, and that the answer of a slave agent is therefore final and cannot be changed during the entire computation. This limitation has been further addressed in [4], where asked agents may provide disjunctive answers to a query at different times, and may also change the answers they have sent previously. In this context, a dynamic iterative belief revision mechanism has been deployed to handle chain reactions of belief revisions among agents involved in a computational process.

Among the operational models proposed for speculative computation [1,2,3,4,6], the one in [4] is the most complex but also the most powerful. A practical implementation for it is very much desired, not only for proof-of-context testing and benchmark investigation, but also for discovering further improvements and/or extensions of the model. The contribution of this paper is to provide the first multi-threaded implementation of a multi-agent system for speculative disjunctive constraint processing. The system allows the master agent to performs speculative computation locally (using multi-threading or-parallelism), and to ask constraint queries to the slave agents. The speculative master agent is associated with one manager thread (MT) and a set of worker threads (WT). The description of the implementation given in the paper re-organises the operational model proposed in [4] to distinguish the tasks of the MT and WTs. A concurrency control mechanism has been introduced to maximise the concurrent execution of the MT and WTs. This implementation design is shown to be good enough to allow for future extensions of the speculative framework to, for instance, hierarchical multi-agent systems.

The paper is organised as follows. Section 2 briefly reviews the operational model of speculative constraint processing presented in [4]. Section 3 describes the multi-threaded implementation in details, as well as the solutions to several concurrent computation issues. Section 4 compares the implementation to the pseudo-parallel approach, and suggests a hybrid-implementation for situations where computational resources (for multi-threading) are limited. Finally, conclusion and future work are given in Section 5.

2 Speculative Disjunctive Constraint Processing

In this section we review the framework of speculative constraint processing and its operational model that has been proposed in [4].

2.1 Speculative Constraint Processing Framework

Definition 1. *Let Σ be a finite set of constants. We call an element in Σ a* slave agent identifier. *An* atom *is of the form either $p(t_1, ..., t_n)$ or $p(t_1, ..., t_n)@S$, where p is a predicate, $t_i (1 \leq i \leq n)$ is a term, and S is in Σ.*

We call an atom with an agent identifier an "*askable atom*", and an atom without an identifier a "*non-askable atom*".

Definition 2. *A* framework for speculative constraint computation, in a master-slave system, *is a triple $\langle \Sigma, \Delta, \mathcal{P} \rangle$, where:*

- *Σ is a finite set of constants;*
- *Δ is a set of rules of the following form, called* default rules *w.r.t. $Q@S$:*

$$Q@S \leftarrow C \| ,$$

  where $Q@S$ is an askable atom, each of whose arguments is a variable, and C is a set of constraints, called default constraints for *$Q@S$;*
- *$\mathcal{P}$ is a constraint logic program, that is, a set of rules R of the form:*

$$H \leftarrow C \| B_1, B_2, ..., B_n ,$$

  where:
  - *H is a non-askable atom; we refer to H as the* head *of R, denoted as $head(R)$;*
  - *C is a set of constraints, called the* constraints *of R, and denoted as $const(R)$;*
  - *each B_i of $B_1, ..., B_n$ is either an askable atom or a non-askable atom, and we refer to $B_1, ..., B_n$ as the* body *of R denoted as $body(R)$.*

For the semantics of the above framework, we index the semantics of a constraint logic program by a *reply set*, which specifies a reply for an askable atom.

Definition 3. *A* reply set *is a set of rules in the form:*

$$Q@S \leftarrow C \| ,$$

where $Q@S$ is an askable atom, each of whose arguments is a variable, and C is a constraint over these variables.

Let $\langle \Sigma, \Delta, \mathcal{P} \rangle$ be a framework for speculative constraint computation, and $\mathcal{R}$ be a reply set. A belief state *w.r.t. $\mathcal{R}$ and Δ is a reply set defined as:*

$$\mathcal{R} \cup \{ \text{“}Q@S \leftarrow C \| \text{”} \in \Delta \mid \neg \exists\ C' \ s.t. \ \text{“}Q@S \leftarrow C' \| \text{”} \in \mathcal{R} \}$$

and denoted as $BEL(\mathcal{R}, \Delta)$.

We introduce the above belief state since, if the answer is not returned, we use a default rule for an unreplied askable atom.

Definition 4. *A goal is of the form* $\leftarrow C \| B_1, ..., B_n$*, where* C *is a set of constraints and the* B_i*'s are atoms. We call* C *the* constraint of the goal *and* $B_1, ..., B_n$ *the* body of the goal.

Definition 5. *A reduction of a goal* $\leftarrow C \| B_1, ..., B_n$ *w.r.t. a constraint logic program* $\mathcal{P}$*, a reply set* $\mathcal{R}$*, and an atom* B_i*, is a goal* $\leftarrow C' \| B'$ *such that:*

- *there is a rule* R *in* $\mathcal{P} \cup \mathcal{R}$ *s.t.* $C \wedge (B_i = head(R)) \wedge const(R)$ *is consistent*[1].
- $C' = C \wedge (B_i = head(R)) \wedge const(R)$
- $B' = \{B_1, ...B_{i-1}, B_{i+1}, ..., B_n\} \cup body(R)$

Definition 6. *A derivation of a goal* $G = \leftarrow C \| Bs$ *w.r.t. a framework for speculative constraint computation* $\mathcal{F} = \langle \Sigma, \Delta, \mathcal{P} \rangle$ *and a reply set* $\mathcal{R}$ *is a sequence of reductions "*$\leftarrow C \| Bs$*",..., "*$\leftarrow C' \| \emptyset$*"*[2] *w.r.t.* $\mathcal{P}$ *and* $BEL(\mathcal{R}, \Delta)$*, where in each reduction step, an atom in the body of the goal in each step is selected.* C' *is called an* answer constraint *w.r.t.* G*,* $\mathcal{F}$*, and* $\mathcal{R}$*. We call a set of all answer constraints w.r.t.* G*,* $\mathcal{F}$*, and* $\mathcal{R}$ *the* semantics of G *w.r.t.* $\mathcal{F}$ *and* $\mathcal{R}$*.*

We refer the readers to [4] for a hotel room reservation example.

2.2 The Operational Model

We briefly describe the execution of the speculative framework. The detailed description can be found in [4]. The execution is based on two phases: a *process reduction phase* and a *fact arrival phase*. The process reduction phase is a normal execution of a program in a master agent, and the fact arrival phase is an interruption phase when an answer arrives from a slave agent.

Figures 1–4 intuitively explain how processes are updated according to askable atoms. In the tree, each node represents a process, but we only show constraints associated with the process. The top node represents a constraint for the original process, and the other nodes represent added constraints for the reduced processes. Let us note that we specify *true* for non-top nodes without added constraints, since the addition of the *true* constraint does not influence the solutions of existing constraints. The leaves of the process tree represent the current processes. Processes that are not in the leaves are deleted processes.

Figure 1 shows a situation of the processes represented as a tree when an askable atom, whose reply has not yet arrived, is executed in the process reduction phase. In this case, the current process, represented by the processed constraints C, is split into two different kinds of processes: the first one is a process using default information, C_d, and is called *default process* [3]; and the other one is the current process C itself, called *original process*, suspended at this point.

When, after some reduction of the default processes (represented in Fig. 2 by dashed lines), the first answer comes from a slave agent, expressing constraint

[1] A notation $B_i = head(R)$ represents a conjunction of constraints equating the arguments of atoms B_i and $head(R)$.

[2] $\emptyset$ denotes an empty goal.

[3] In this figure, we assume that there is only one default for brevity.

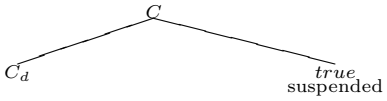


Fig. 1. When $Q@S$ is processed in process reduction phase

C_f for this askable literal, we update the default processes as well as the original suspended process as follows:

- Default processes are reduced to two different kinds of processes: the first kind is a process adding C_f to the problem to solve, and the other is the current process itself which is suspended at this point.
- The original process is reduced to two different kinds of processes as well: the first kind is a process adding $\neg C_d \wedge C_f$, and the other is the original process, suspended at this point.

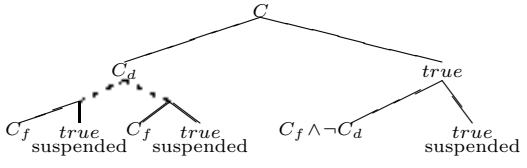


Fig. 2. When the first answer C_f for $Q@S$ arrives

Let $\leftarrow C \| Bs$ be a goal containing $Q@S$. Suppose that it is reduced into $\leftarrow C \wedge C_d \| Bs \backslash \{Q@S\}$ by a default rule "$Q@S \leftarrow C_d \|$". To retain the previous computation as much as possible, we process the query by the following execution:

1. We add C_f to the constraint of every goal derived from the default process.
2. In addition to the above computation, we also start computing a new goal:

$$\leftarrow C \wedge \neg C_d \wedge C_f \| Bs \backslash \{Q@S\}$$

to guarantee completeness.

When an alternative answer, with the constraint C_a, comes from a slave agent (Fig. 3), we need to follow the same procedure as when the first answer comes (Fig. 2), except that now the processes handling only default information are suspended. So, this is done by splitting the suspended default process(es), in order to obtain the answer constraints that are logically equivalent to the answer constraints of:

$$\leftarrow C \wedge C_d \wedge C_a \| Bs \backslash \{Q@S\},$$

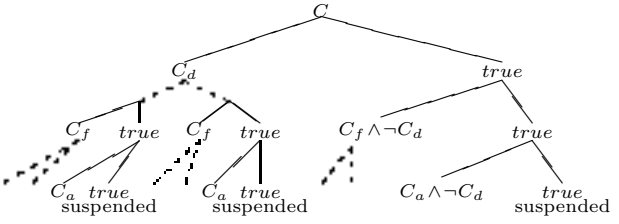


Fig. 3. When the alternative answer C_a for $Q@S$ arrives

as well as by splitting the suspended original process, in order to obtain the answer constraints that are logically equivalent to the answer constraints of $\leftarrow C \wedge \neg C_d \wedge C_a \| Bs \backslash \{Q@S\}$ (Fig. 3). By gathering these answer constraints, we can compute all answer constraints for the alternative reply.

On the other hand, when a revised answer with the constraint C_r arrives, all processes using the first (or current) answer are split, in order to obtain the answer constraints that are logically equivalent to the answer constraints of:

$$\leftarrow C \wedge C_f \wedge C_r \| Bs \backslash \{Q@S\},$$

and the suspended original process is split as well, in order to obtain the answer constraints that are logically equivalent to the answer constraints of $\leftarrow C \wedge \neg C_f \wedge C_r \| Bs \backslash \{Q@S\}$ (Fig. 4). By gathering these answer constraints, we can override the previous reply by the revised reply.

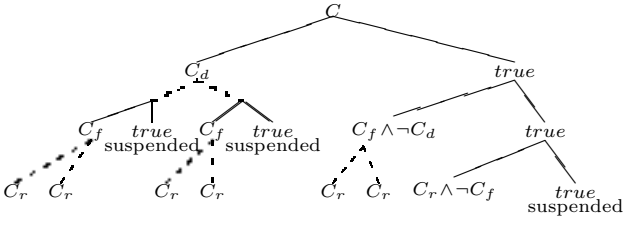


Fig. 4. When the revised answer C_r for $Q@S$ arrives

3 A Multi-threaded Implementation

In [4], the detailed operational model is described as a multi-*processing* computation. There are two types of processes – *finished processes* that represent successfully terminated computational branches, and *ordinary processes* that represent non-terminated branches. An ordinary process can be either an *original process* that is always suspended or an *active process* that searches down an open branch.

In practice the operational model can be implemented in two ways:

1. we represent each *process* as a state, and use a single process/thread to manipulate the states in a pseudo-multi-threading (serialised) fashion. This is very close to the model description;
2. we execute each process using a real thread, so that different (non-suspended) processes can execute concurrently.

The multi-threaded approach avoids overheads caused by state selection and management that the serialised approach has, and allows or-parallelism which will benefit the proof search. However, using one thread for each process may not always be necessary and may cause extra overheads such as in inter-threads communication. For example, original processes are always suspended and can never be resumed, though it may spawn new processes that are not suspended. Preferably they should be managed as states instead, for easy update when a relevant answer is returned. This is also true for finished process. In this section, we describe a practical implementation for the operational model, which considers various efficiency aspects.

3.1 Overview

The model is implemented as a *speculative computation module*, and we refer to it as a *speculative agent*. A set of agents (some of them may not be speculative agents) can be deployed to one or more host machines on a network. Agents interact with each other via messages (containing queries or answers). Since the operational model proposed in [4] is for simple master-slave systems only, in this paper we also assume that there can be only one master, i.e. the only speculative agent, in the set of deployed agents, and the rest are the slaves. The master can send queries to the slaves, but a slave cannot send queries to the master or other slaves. Hence, only the master can perform constraint processing with iterative revision for disjunctive answers. But bear in mind that our implementation is in fact designed in a way that it can be easily extended for hierarchical multi-agent systems similar to that defined in [2].

As illustrated in Fig. 5, each agent has the following internal components:

Communication Interface Module (COM): this is the only interface for inter-agent communications. It accepts queries or answers sent by the agent's master or slaves, and forwards the agent's answers or queries to the master or the appropriate slaves. The *reception list* and the *address book* are used for keeping track of the queries received and the master/slave addresses[4].

Speculative Computation Unit (SCU): this is the central processing unit of the agent that performs speculative computations for one or more queries.

Default Store (Δ) and Program ($\mathcal{P}$): they are self-explained, and form the *static knowledge* of the agent.

Answer Entry, Choice Point and Finish Point Stores (AES, CPS, FPS): AES stores the *answer entries* that are created from either Δ or the returned

[4] Both these features will be essential when the implementation is extended for hierarchical multi-agent systems.

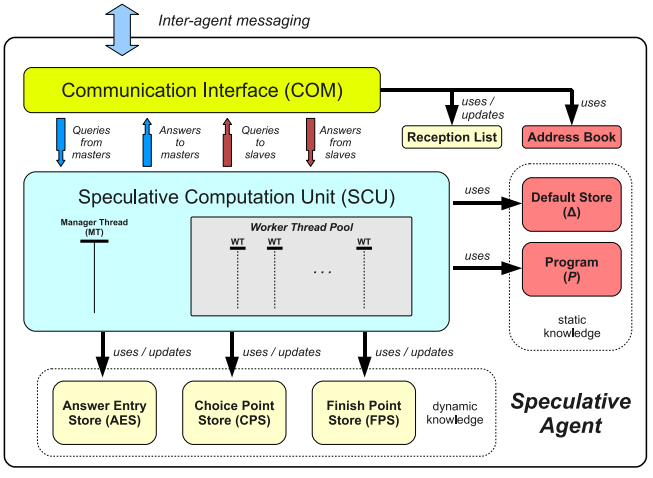


Fig. 5. Agent Internal Components

answers from the slaves (i.e. the reply set $\mathcal{R}$). CPS stores the computation choice points (CP), each of which represents the state of a (suspended) original process. FPS stores the finish points (FP), which contain the results of finished processes. The three stores are used by SCU and form the *dynamic knowledge* of the agent.

In the following sections, we describe how these components are implemented.

3.2 Implementing the Communication Interface Module (COM)

Agents communicate asynchronously via messages sent over TCP connections. Each agent on the network is uniquely identified by a *socket* of the form `IP:Port`, where `IP` is the network address of the agent's host and `Port` is the port number reserved for the agent on the host. Therefore, several agents may run simultaneously on a host.

During the design of an agent's program, the sockets for the slaves may not be known, or they may be changed during agent deployment. Therefore, each agent uses aliases to identify its slaves locally. For example, in an askable atom $Q@S$ appearing in $\mathcal{P}$ or Δ, S is the alias of a slave. The address book stores the mapping between the slave aliases and the slave sockets, and it can be generated/updated during agent (re-)deployment.

There are two types of messages for inter-agent communications:

- a *query message* of the form `query(From, Q@S, Cmd)`, where `From` is the socket of the sender, `Q` is a query, `S` is the recipient's alias used by the sender, and `Cmd` is a command of either `start` or `stop`. If the command is `start`, it indicates a request for the recipient (i.e. the slave) to start a computation for the query; otherwise if the command is `stop`, it asks the recipient to stop

the computation for a query previously requested and to free the resources. The "stop" signal (in this paper) is merely used for the execution control of the agent.
- an *answer message* of the form `answer(From, Q@S, ID, Ans)`, where `From`, `Q` and `S` are described as above, `Ans` is a set of constraints as the answer to the query, and `ID` is the answer identifier by the sender and is used to distinguish between a *revised answer* and an *alternative answer*.

COM waits for any incoming message and handles it as follows:

- if it is an inter-agent message `query(Master, Q@S, start)` from the agent's socket, COM creates an entry `<RID, Q@S, Master>` in the reception list, where `RID` is a new query entry ID, and then sends a message `start(RID, Q@S)` to the *manager thread* (MT) in SPU (to be described soon);
- if it is an inter-agent message `query(Master, Q@S, stop)`, COM removes the entry `<RID, Q@S, Master>` from the reception list, and then sends a message `stop(RID)` to MT;
- if it is an inter-agent message `answer(Slave, Q@S, ID, Ans)`, COM simply forwards it as `answer(Q@S, ID, Ans)` to MT;
- if it is an internal message `answer(RID, Q, ID, Ans)` from MT or from one of the *worker threads* (WT) in SPU, COM looks up `<RID, Q@S, Master>` from the reception list, and then sends the inter-agent message `answer(Self, Q@S, ID, Ans)` to the master, where `Self` is the current agent's socket;
- if it is an internal message `query(Q@S)` from a WT, COM looks up the slave's socket from the address book using `S`, and then sends the inter-agent message `query(Self, Q@S, start)` to the slave.

3.3 Implementing the Speculative Computation Unit (SCU)

SCU can be seen as a collection of concurrent threads. Specifically, there is a persistent *manager thread* (MT) and zero or more *worker threads* (WT). MT is responsible for updating/revising the choice points/finish points and for spawning new WT(s) when a new query or answer is received, and WTs are responsible for constraint processing.

The three stores AES, CPS and FPS are used and maintained by both MT and WTs. AES stores three types of answer entries (AE), all of which have the form `<AID, Q@S, Type, Ans>`, where `AID` is the entry ID, `Q@S` is the query and the slave alias, `Type` is the entry's type and `Ans` is the set of constraints associated with the entry:

- If `Type` is `so`, then this is a *speculative original answer entry*, and `Ans` is equal to the conjunction of the negations of all the defaults in Δ for `Q@S`[5] if there is any default, and is equal to `true` otherwise;
- if `Type` is `nso`, then this is a *non-speculative original answer entry* and `Ans` is `true`;

[5] i.e. $\bigwedge_{(Q@S \leftarrow C_d \|) \in \Delta} \neg C_d$.

- If `Type` is `d`, then this is a *default answer entry*, and `Ans` is equal to a corresponding default answer for `Q@S` in Δ;
- otherwise, `Type` is `r(ID)` and this is an *ordinary answer entry*, where `ID` and `Ans` are from an answer returned by the slave `S` for `Q`.

CPS stores the states of original processes (or called *choice points* (CP)), each of which has the form `<QID, PID, G, C, WA, AA>`, where `QID` is the (top level) query and its ID, `PID` is the process ID, `G` and `C` are the set of remaining sub-goals and the set of constraints collected so far respectively, `WA` and `AA` are the set of awaiting answer entries and the set of assumed answer entries respectively. `QID` is used by a process to "remember" what query its computation is for, and hence has two components `(RID-`$\mathtt{Q}_{top}$`)`, where `RID` is the reception entry ID, and $\mathtt{Q}_{top}$ is the initial query for the process. It is necessary to record $\mathtt{Q}_{top}$ so that when a process finishes successfully (i.e. `G` becomes empty), the variable bindings between the answer (i.e. set of constraints) and the initial query can be preserved. Each element in `WA` and `AA` has the form `(AID, Q@S)`, where `AID` is the ID of an answer entry that the process is awaiting or is assuming for the sub-goal `Q@S`. Note that it is also necessary to record `Q@S` here despite having already recorded `AID`, because if later an assumed answer needs to be revised, the correct variable bindings between the query sent (to the slave) and the answer returned (from the slave) can be obtained.

FPS stores the states of finished processes (or called *finish points* (FP)), each of which has the form `<QID, PID, C, AA>`, where `QID`, `PID` and `AA` are as described above, and `C` is the final set of constraints collected, i.e. the answer, already sent to the master for the query associated with `QID`.

Each WT represents an active process, and its state can be represented as `<QID, PID, G, C, AA>`. It is just like a CP except that it does not have the awaiting answer entry set (i.e. no `WA`).

It is also important to keep track of what AE is currently assumed/awaited by what WTs, CPs and FPs. Such usages of AE are recorded as *subscriptions* in a *directory* as a part of AES. Each subscription has the form `sub(AID, PID)`, where `AID` is the answer entry ID and PID is the ID of a WT, CP or FP.

3.4 The Execution of the Manager Thread and the Worker Threads

The multi-threaded operational model is based on the pseudo-parallel (serialised) operational model proposed in [4], but with improved "process management" allowing true or-parallelism during the computation:

- In the serialised model, the computation interleaves with the *process reduction phase* and the *fact arrival phase*. When it enters the process reduction phase, one active process is selected at a time for resolving a sub-goal. In the multi-threaded model, each WT can enter the process reduction phase and resolve sub-goals independently and concurrently to others. No process selection is required.
- In the serialised model, when it enters the fact arrival phase, all the relevant processes (active or suspended) are updated, and necessary new processes

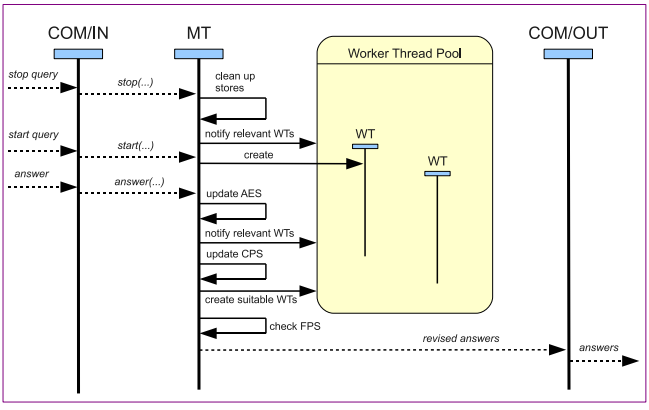
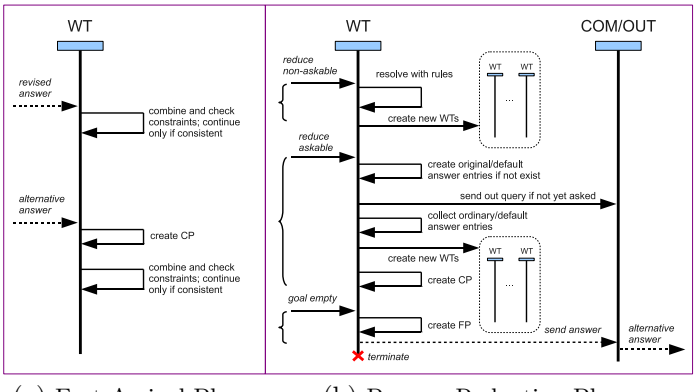


Fig. 6. Execution of MT



(a) Fact Arrival Phase (b) Process Reduction Phase

Fig. 7. Execution of WT

from original processes are created at the same time. In the multi-threaded model, the fact arrival phase is splitted and is done by the MT and WTs separately. The MT is responsible for revising the answers from existing finished processes (i.e. the finish points), updating original processes (i.e. the choice points) and creating appropriate new WTs from choice points. The MT also notifies relevant WTs about the newly returned answer via messaging, but will not change the state of WTs directly. On the other hand, when a WT receives such notification from MT, it will check for consistency of the new answer independently from others, and create new choice point if needed (e.g. in the case where it is assuming a default answer and an alternative answer is received). Different WTs can update themselves concurrently.

The key execution steps for MT and WT are illustrated in Figure 6 and Figure 7. The detailed descriptions are provided in Appendix A.1 and Appendix A.2.

3.5 Resolving Concurrency Issues

Inside SPU, MT and WTs execute concurrently, and they all require read/write access to the three stores AES, CPS and FPS. Potential conflicts between MT and a WT, or between WTs may arise. Firstly, it is possible that after a WT spawns several children WTs, and just before it can make all the answer entry subscriptions for the children, MT receives an answer and notifies only some of its children (e.g. the subscription process is not yet complete). Secondly when two WTs encounter the same askable atom at the same time, and if there is no original answer entry for that atom yet, then the original answer entry may be created twice and the query may be sent twice by the two WTs. Hence, the three stores are considered as "critical regions" and need to be protected. One naïve solution is to make all the iteration steps performed by WT or MT atomic. But this will greatly reduce the chance for concurrent processing and hence remove almost all the benefits brought by the multi-threaded implementation. Therefore, "fine grained" atomicity control is needed for the executions of MT and WTs.

Let's consider the first problem. The potential conflict is between MT and WT, and is not between WTs. Although several WTs may need to update the subscriptions in the directory of AES, they only modify the ones associated with their IDs or with their new born children's IDs. As long as the children WTs do not start working until their parent WT has made all the correct subscriptions for them, there won't be any conflict. Also, WTs can only create new choice points in CPS and create new finish points in FPS according to their own states, there is no potential conflict of updating CPS and FPS either. Therefore, the execution of a MT's message handling step cannot (safely) interleave with that of the process reduction step or the fact arrival step of any WT, but the executions of WTs' steps can interleave without problems. To impose such control, we have introduced an atomic counter[6] called the "busy worker counter" (BC). Whenever a WT starts to perform a fact arrival step or reduction step, it will increment BC; and whenever it finishes one step, it will decrement BC. We also introduce an atomic flag called the "waiting/working manager flag"(WF). Whenever MT receives an answer, it will *set* WF to `1`; and when MT finishes handling one returned answer, it will *clear* WF to `0`. The safe exclusive execution control between MT and WTs using BC and WF are as follows[7],

WT's Execution Cycle	**MT's Execution Cycle**
1. (atomic step) waits for WF to be cleared and then increments BC; 2. *performs either fact arrival step or reduction step;* 3. decrements BC	1. waits for a returned answer; 2. sets WF 3. waits for BC to reach `0`; 4. *handles returned answer;* 5. clears WF

[6] I.e. its value update is atomic.

[7] Pseudo-code in Prolog is provided in Appendix B.

Hence, whenever a WT performing a fact arrival step or process reduction step, MT is not allowed to process any received answer; whenever MT has an answer waiting to be processed or being processed, no WT can perform a new step.

Let's now consider the second problem. The potential conflict is between two WTs when they both try to collect/create answer entries for an askable goal. The solution is relatively easy: we have introduced a mutex M_{AES} and control the WT's execution as follows,

When a WT tries to collect answer entries for `Q@S`:

- if an original answer entry for `Q@S` exists in AES, *continues as normal*;
- otherwise, (1) locks M_{AES}; (2) if AES still does not contain an original answer entry for `Q@S`, then *creates the original and default answer entries, and then sends out the query*; (3) unlocks M_{AES}.

The operation of locking a mutex succeeds immediately if the mutex has not been locked by any other thread yet; otherwise it causes the current thread to be suspended. The suspended thread is revived only when the mutex is unlocked, and then the revived thread tries again to lock the mutex. In the above example, it is possible that while a thread is waiting to lock M_{AES}, the thread already locking M_{AES} creates the answer entries. Therefore, in Step 2 checking again whether an original answer entry exists is necessary.

4 Discussions

The described mutli-threaded implementation is implemented in YAP Prolog [7]. We chose YAP not only because it has the necessary CLP and multi-threading supports, but also because it is considered as the one of the fastest Prolog engines that is free and open source.

We have tested the implementation with meeting scheduling examples described in [4] but with increased size. During the testing, we used YAP's default maximum number of WTs of 100 and were able to compute the correct answers within the order of 1 second. For large problems, e.g. if a query would lead to more than 10 (non-askable) sub-goals, each with more than 10 rules with constraints that are always consistent, the number of WTs would exceed 100. Our implementation is able to cope with such problems by setting a higher WT number limit, e.g. 1000, at the expense of initial memory consumed by YAP[8].

In practice, to strike a balance between the number of WTs and the memory consumption, our implementation can be adapted to use a *hybrid* approach, which would implement two types of WTs: *normal workers* and *super worker*. A *normal worker* would execute as an active process as described in the multi-threaded model. A *super worker* would behave like the serialised model [4] and manage several processes in a round-robin fashion. In this way, memory consumption would be reduced whilst maintaining the effect of a high number of

[8] 100 maximum threads in YAP require about 2MB memory, 1000 threads require about 4MB and 9999 threads require about 109MB.

WTs. For example, let M be the maximum number of WTs that an agent's SPU can have, then there can be $M-1$ (at most) normal workers and 1 super worker. During the computation, when there are N ($N > M-1$) active processes, $M-1$ of them are handled by the normal workers, and the rest of them are handled by the super worker. When an active process terminates (either due to failure or finish), the normal worker can release it and acquire another active process state from the super worker to continue.

5 Conclusion

In this paper, we have presented a practical multi-threaded implementation for speculative constraint processing with iterative revision for disjunctive answers, and suggested a hybrid implementation for situation where multi-threading support is limited by resource constraint. Although the implementations are based on the operational model described in [4], which is for simple master-slave systems where only the master can perform speculative computation, they are designed to be extendable for hierarchical master-slave systems. As a future work, we will prove the correctness of an extended operational model for a hierarchy of master-slave agents and extend the current implementation to support this more general type of multi-agent systems. We will also perform benchmarking of the system with large examples, and apply it in real world applications, such as planning and online booking systems.

Acknowledgment

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Japan, Grant-in-Aid for Scientific Research (B), 19300053, and is continuing through participation in the International Technology Alliance sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence.

References

1. Satoh, K., Inoue, K., Iwanuma, K., Sakama, C.: Speculative computation by abduction under incomplete communication environments. In: ICMAS, pp. 263–270 (2000)
2. Satoh, K., Yamamoto, K.: Speculative computation with multi-agent belief revision. In: AAMAS, pp. 897–904 (2002)
3. Satoh, K., Codognet, P., Hosobe, H.: Speculative constraint processing in multi-agent systems. In: Lee, J.-H., Barley, M.W. (eds.) PRIMA 2003. LNCS (LNAI), vol. 2891, pp. 133–144. Springer, Heidelberg (2003)
4. Ceberio, M., Hosobe, H., Satoh, K.: Speculative constraint processing with iterative revision for disjunctive answers. In: Toni, F., Torroni, P. (eds.) CLIMA 2005. LNCS (LNAI), vol. 3900, pp. 340–357. Springer, Heidelberg (2006)
5. Satoh, K.: Speculative computation and abduction for an autonomous agent. IEICE Transactions 88-D(9), 2031–2038 (2005)

6. Inoue, K., Kawaguchi, S., Haneda, H.: Controlling speculative computation in multi-agent environments. In: Proc. Second Int. Workshop on Computational Logic in Multiagent Systems (CLIMA 2001), pp. 9–18 (2001)
7. YAP Prolog 5.1.3 manual (June 2008), `http://www.dcc.fc.up.pt/~vsc/Yap/index.html`

A Execution Description of MT and WT

A.1 Execution of MT (Illustrated in Fig. 6)

MT processes each message it receives from COM:

- if the message is $start(RID, Q)$, it spawns a new WT with initial state $\langle QID, PID_{new}, Q, \emptyset^{9}, \emptyset^{10}\rangle$, where $QID = (RID, Q)$, PID_{new} is a new process ID.
- if the message is $stop(RID)$, then
  1. it removes all the choice points in CPS and all the finish points in FPS that are associated with RID;
  2. it broadcasts a message $stop(RID)$ to all the WTs;
- if the message is $answer(Q@S, ID, C_{new})$:
  - if there exists an answer entry $\langle AID, Q@S, r(ID), C_{old}\rangle$ in AES, then **the received answer is a revised answer** (following Fig. 4):
    1. MT updates the existing answer entry to be $\langle AID, Q@S, r(ID), C_{new}\rangle$;
    2. for each WT subscribing AID, MT sends a message $rev(AID, Q@S, C_{new})$ to the WT (so that the WT can check C_{new} for consistency);
    3. for each FP of $\langle QID, PID, C_{final}, AA\rangle$ that is subscribing AID and $QID = (RID, Q_{top})$, if $C_{final} \neq C_{final} \wedge C_{new}$, then MT sends a message $answer(RID, Q_{top}, PID, C_{final} \wedge C_{new})$ to COM;
    4. for each CP of $\langle QID, PID, G, C, WA, AA\rangle$ that is subscribing AID, if $C_{all} = C \wedge C_{new}$ is consistent, then MT updates it to be $\langle QID, PID, G, C_{all}, WA, AA\rangle$; otherwise, MT removes the CP and the CP's subscriptions;
    5. let $\langle AID_o, Q@S, O, C_o\rangle$ be an original answer entry for $Q@S$, where O is either *so* or *nso*, for each choice point of $\langle QID, PID, G, C, WA, AA\rangle$ that is subscribing AID_o and $C_{all} = C \wedge \neg C_{old} \wedge C_{new}$ is consistent:
       - if WA contains only $(AID_o, Q@S)$, then MT creates a new WT with $\langle QID, PID_{new}, G, C_{all}, AA \cup \{(AID, Q@S)\}\rangle$, and subscribes all the answer entries in AA and that with AID for the new WT (i.e. for each $(AID', Q'@S') \in AA \cup \{(AID, Q@S)\}$, it adds $sub(AID', PID_{new})$ to the directory in AES);
       - otherwise, MT creates a new CP of $\langle QID, PID_{new}, G, C_{all}, WA \setminus \{(AID_o, Q@S)\}, AA \cup \{(AID, Q@S)\}\rangle$ in AES, and subscribes all the answer entries in AA and in WA for the new CP;
  - **otherwise, it is a first/alternative answer** (following Fig. 2 and Fig. 3):
    1. MT creates a new answer entry $\langle AID_{new}, Q@S, r(ID), C_{new}\rangle$ in AES;
    2. for each default answer entry $\langle AID_d, Q@S, d, C_d\rangle$ in AES:

[9] This is the initially empty set of constraints.
[10] This is the initially empty set of assumed answer entries.

- for each WT subscribing AID_d, MT sends a message $alt(AID_{new}, AID_d, Q@S, C_{new})$ to it;
- for each FP of $\langle QID, PID, C_{final}, AA\rangle$ that is subscribing AID_d and $QID = (RID, Q_{top})$, if $C_{final} \neq C_{final} \wedge C_{new}$, then MT sends a message $answer(RID, Q_{top}, PID, C_{final} \wedge C_{new})$ to COM;
- for each CP of $\langle QID, PID, G, C, WA, AA\rangle$ that is subscribing AID_d,
  (a) MT updates the CP to be $\langle QID, PID_{new}, G, C, WA \cup \{(AID_d, Q@S)\}, AA \setminus \{(AID_d, Q@S)\}\rangle$;
  (b) if $C_{all} = C \wedge C_{new}$ is consistent, then
    · if WA contains only $(AID_d, Q@S)$, then MT creates a new WT with $\langle QID, PID_{new}, G, C_{all}, AA \cup \{(AID, Q@S)\}\rangle$, and subscribes all the answer entries in AA and that with AID for the new WT;
    · otherwise, MT creates a new CP of $\langle QID, PID_{new}, G, C_{all}, WA \setminus \{(AID_d, Q@S)\}, AA \cup \{(AID, Q@S)\} \setminus \{(AID_d, Q@S)\}\rangle$ in AES, and subscribes all the answer entries in $AA \cup WA \cup \{(AID, Q@S)\} \setminus \{(AID_d, Q@S)\}$ for the new CP;

3. let $\langle AID_o, Q@S, O, C_o\rangle$ be an original answer entry for $Q@S$, where O is *so* or *nso*, for each choice point of $\langle QID, PID, G, C, WA, AA\rangle$ that is subscribing AID_o and $C_{all} = C \wedge C_o \wedge C_{new}$ is consistent:
   - if WA contains only $(AID_o, Q@S)$, then MT creates a new WT with $\langle QID, PID_{new}, G, C_{all}, AA \cup \{(AID, Q@S)\}\rangle$, and subscribes all the answer entries in AA and that with AID for the new WT;
   - otherwise, MT creates a new CP of $\langle QID, PID_{new}, G, C_{all}, WA \setminus \{(AID_o, Q@S)\}, AA \cup \{(AID, Q@S)\}\rangle$ in AES, and subscribes all the answer entries in $AA \cup WA \cup \{(AID, Q@S)\} \setminus \{(AID_o, Q@S)\}$ for the new CP;

A.2 Execution of WT (Illustrated in Fig. 7)

The execution of a WT can be seen as a loop with the following steps performed at each iteration (let its initial state at each iteration be $\langle QID, PID, G, C, AA\rangle$):

– If there is an internal message received by the WT (i.e. from MT), it enters the **Fact Arrival Phase**:
  - if the message is $rev(AID, Q@S, C_r)$ where $(AID, Q@S) \in AA$ (see Fig. 4), let $C_{all} = C \wedge C_r$: if C_{all} is consistent, then the WT continues with $\langle QID, PID, G, C_{all}, AA\rangle$. Otherwise, the WT removes all of its subscriptions in AES and terminates;
  - if the message is $alt(AID_a, AID_d, Q@S, C_a)$ where AID_d is an ID of a default answer entry (following Fig. 2),
    1. it creates a new CP of $\langle QID, PID_{new}, G, C, \{(AID_d, Q@S)\}, AA \setminus \{(AID_d, Q@S)\}\rangle$ in CPS, and subscribes for all the answer entries in AA for the new CP;
    2. if $C_{all} = C \wedge C_a$ is consistent, then the WT continues with $\langle QID, PID, G, C_{all}, AA \cup \{(AID_a, Q@S)\} \setminus \{(AID_d, Q@S)\}\rangle$. Otherwise, it removes all of its subscriptions and terminates;

- if the message is $stop(RID)$, and RID is equal to the query ID in QID, then the WT removes all of its subscriptions and terminates;

– Otherwise, it enters the **Process Reduction Phase** and *tries to* select L from G:
  - if G is empty and thus no L can be selected, the current computation succeeds:
    1. let $QID = (RID, Q_{top})$, the current WT sends a message $answer(RID, Q_{top}, PID, C)$ to COM;
    2. it creates a FP of $\langle QID, PID, C, AA\rangle$ and then terminates. Note that it does not need to make answer entry subscriptions for the new FP or to remove its subscriptions, because the new FP "inherits" them.
  - if L is not an askable atom, for every rule R such that $C_{new} = C \wedge (L = head(R)) \wedge const(R)$ is consistent, the current WT spawns a new WT with state $\langle QID, PID_{new}, G \setminus \{L\} \cup body(R), C_{new}, AA\rangle$ and subscribes all the answer entries in AA for the new WT. Then the current WT removes all of its subscriptions and terminates[11].
  - if L is an askable atom $Q@S$ (where S must be ground): if there exists $(AID, Q'@S) \in AA$ such that Q and Q' are identical (i.e. they are not variants), then the WT continues with $\langle QID, PID, G \setminus \{L\}, C, AA\rangle$[12]. Otherwise (following Fig. 1),
    1. it collects $(AIDo, AIDS)$ from AES as follows:
       - if there exists some ordinary answer entries for $Q@S$, let $AIDo$ be the non-speculative original answer entry ID for $Q@S$, and $AIDS$ be the set of ordinary answer entry IDs, whose associated answer constraints are consistent with C;
       - otherwise,
         (a) if there exists no original answer entry for $Q@S$, then the WT
           i. creates $\langle AID^{so}_{new}, Q@S, so, C_{so}\rangle$ in AES, where C_{so} is the conjunction of the negations of all the default constraints for $Q@S$ in Δ if there is some default constraint, or is $true$ if there is none;
           ii. creates $\langle AID^{nso}_{new}, Q@S, nso, C_{nso}\rangle$ in AES, where C_{nso} is $true$;
           iii. creates a default answer entry $\langle AID^{i}_{new}, Q@S, d, C^i_d\rangle$ for each default constraint C^i_d for $Q@S$ in Δ;
           iv. sends a message $query(Q@S)$ to COM;
         (b) let $AIDo$ be AID^{so}_{new}, and $AIDS$ be the set of default answer entry IDs, whose associated answer constraints are consistent with C;
    2. for each answer entry $\langle AID, Q@S, Type, C_a\rangle$ such that $AID \in AIDS$, the current WT spawns a new WT with state $\langle QID, PID_{new}, G \setminus \{Q@S\}, C \wedge C_a, AA \cup \{(AID, Q@S)\}\rangle$ and subscribes all the answer entries in $AA \cup \{(AID, Q@S)\}$ for the new WT;
    3. the current WT creates a new CP of $\langle QID, PID_{new}, G \setminus \{Q@S\}, C, \{(AIDo, Q@S)\}, AA\rangle$ in CPS, and subscribes all the answer entries in AA plus that with $AIDo$ for the new CP;
    4. the current WT removes all of its subscriptions and terminates[13].

[11] As an optimisation, if there are $N > 0$ possible new processes (states), then only $N - 1$ new WTs are spawned, and the current WT continues as Nth process.

[12] This is an optimisation to the original operational model, which prevents unnecessary new processes (threads) to be created.

[13] Optimisation similar to footnote 11 can be applied.

B Pseudo-Code for the Implementation of Exclusive Control between the Manager Thread and Worker Threads

YAP Prolog only provides *message queues* and *mutexes* for multi-threading support [7].

```
% "m_bc" and "m_wf" are the
    mutexes for BC and WF;
% "v_bc" is the counter for BC
% "mq_bc" is the message queue
    for notifications about BC

% for WT
wt_loop :-
  mutex_lock(m_wf),
  mutex_lock(m_bc),
  mutex_unlock(m_wf),
  increment(v_bc),
  mutex_unlock(m_bc),
  // process reduction or fact
      arrival step
  mutex_lock(m_bc),
  decrement(v_bc),
  (v_bc(V),  V == 0 ->
    send_notification_to(mq_bc
        )
  ;
    true
  ),
  mutex_unlock(m_bc),
  wt_loop.
```

```
% for MT
mt_loop :-
  // wait for received answer,
  mutex_lock(m_wf),
  wait_for_zero_bc,
  // handle received answer
  mutex_unlock(m_wf),
  mt_loop.

wait_for_zero_bc :-
  mutex_lock(m_bc),
  clear_any_notification_in(mq_bc),
  (v_bc(V), V > 0 ->
    mutex_unlock(m_bc),
    wait_for_notification_in(mq_bc),
    wait_for_zero_bc
  ;
    mutex_unlock(m_bc)
  ).
```