Belief Caching in 2APL

Mehdi Dastani and Marc van Zee

Utrecht University
Department of Information and Computing Science
P.O. Box 80.089, 3508 TB Utrecht
The Netherlands

Abstract. The BDI-oriented multi-agent programming language 2APL allows the implementation of an agent's beliefs in terms of logical facts and rules. An agent's beliefs represent information about the agent's surrounding environment including other agents. Repeated querying of the beliefs by the 2APL interpreter causes unnecessary overhead resulting in poor run-time performance of the interpreter. We propose a heuristic to reduce the number of such queries by using belief caching. We show that our belief caching implements and extends an existing caching proposal. Benchmarking results indicate that belief caching can lead to significant improvements.

1 Introduction

The multi-agent programming language 2APL[1] supports the implementation of individual agents that can perform high-level reasoning and deliberation about their information (i.e., beliefs) and objectives (i.e., goals to achieve) in order to decide what actions to perform [3]. Beliefs and goals in 2APL are declarative; Beliefs are represented by a set of Horn clauses and goals are represented by conjunctions of first-order atoms. While this allows the development of flexible and declarative agent programs, repeated inferencing triggered by queries to the beliefs can result in poor performance. When developing multi-agent systems for time critical applications, performance issues are often a key concern, potentially adversely impacting the adoption of BDI-based agent programming languages and platforms as an implementation technology [1]. For example, if agent programming languages want to provide better support for implementing autonomous robots, one of the requirements is real-time reactivity to events, which is currently lacking [6].

We present an inference method based on caching within the 2APL interpreter that reduces the number of belief queries. Our motivation for this approach is based on the observation that belief queries are responsible for most of the deliberation time within a 2APL deliberation cycle and that most belief queries are redundant because they are being performed repeatedly while relevant parts of the belief base do not change, meaning the result of such queries

[1] For more information, see: http://apapl.sourceforge.net/.

will be the same. Using the notion of caching is therefore likely to be an optimization. Recently, it is shown [1] that it is theoretically possible to improve the run-time execution of BDI-based agent programs by using belief caching. However, this proposal focuses purely on the optimization of belief queries within one so-called *update cycle*, which consists of a *query phase* and an *update phase.* Also, the proposal has not been implemented. Our approach specializes the concept of a general update cycle to an update cycle *for each individual query* that may cover multiple deliberation cycles. We show that the general update cycle is contained in our proposal and that our proposal is more fine-grained leading to an increased number of queries answered by the cached beliefs.

We implement belief caching in the 2APL interpreter by performing a belief query only if the belief base has been updated in a way that is relevant to this query. We exploit the fact that both belief queries and belief updates are static in 2APL, which makes it possible to determine what belief update will change what belief query at compile-time. In order to do this, we define the notion of *relevance* for belief queries by making use of *query dependency sets* in the belief base. We have implemented our belief caching approach into the latest version of 2APL.[2] Additionally, we have implemented a generative benchmarking tool, which allows the reader to test the working of belief caching very easily. The manual for the benchmarking tool can be found in the 2APL manual.

The structure of this paper is as follows. In Section 2 we introduce 2APL together with the parts that are relevant to our analysis. In Section 3, we introduce our belief caching approach, compare it with the abstract performance model as proposed in [1], and show how our approach can be seen as an extension to this work. Finally, we provide implementation details and benchmarking results in Section 4.

2 2APL - A Practical Agent Programming Language

The programming language 2APL is developed to implement multi-agent systems [3]. In 2APL, individual agents are programmed in terms of beliefs, goals, actions, plans, events, and three types of practical reasoning rules. The beliefs and goals of 2APL agents are implemented in a declarative way, while plans are implemented in an imperative style. The declarative part of the programming language supports the implementation of an agent's reasoning task and the update of its mental state. The imperative part of the programming language facilitates the implementation of plans, control flow, and mechanisms such as procedure call, recursion, and interfacing with legacy codes. 2APL agents can perform different types of actions such as belief and goal update actions, belief and goal test actions (belief and goal queries), external actions (including sense actions) and communication actions. The practical reasoning rules can be applied to generate plans. The first type of rules is designed to generate plans for achieving goals (so-called Planning Goal rules, or PG rules), the second to

[2] The sources of the latest 2APL version can be downloaded from http://www.apapl.sourceforge.net/.

process external events, messages and abstract actions (so-called Procedure Call rules, or PC rules), and the third to process internal events for repairing failed plans (so-called Plan Repair rules, or PR rules). Each practical reasoning rule has a belief query that specifies the belief state in which the rule can be applied.

2APL agents are autonomous in the sense that they continuously *deliberate* on their mental states (beliefs, goals and plans) in order to decide which plans to select and execute. This deliberation mechanism, which is an integral part of the 2APL interpreter, iterates over a reasoning cycle, depicted in Figure 1. The reasoning cycle starts by applying applicable PG rules of an agent program in order to generate *plans* to achieve the agent's goals. The reasoning cycle continues by executing the generated plans. Then, the received internal and external events and messages are processed by applying PC and PR rules. We would like to emphasize that the application of all practical reasoning rules as well as the execution of belief test actions require queries to the belief base. The fact that the application of practical reasoning rules is the core activity of each reasoning cycle implies that the belief query actions constitute the most frequent operations in the reasoning cycle. Therefore, any significant reduction in the number of belief queries is expected to improve the performance of the 2APL interpreter.

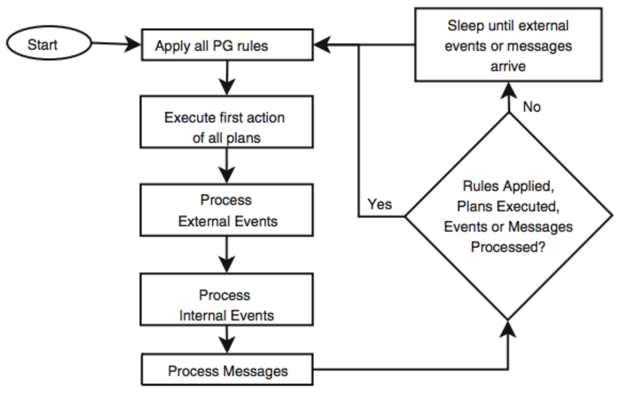

Start
Apply all PG rules
Execute first action of all plans
Process External Events
Process Internal Events
Process Messages
Rules Applied, Plans Executed, Events or Messages Processed?
Yes
No
Sleep until external events or messages arrive

Fig. 1: The 2APL deliberation cycle

2.1 Belief queries

Belief queries can occur at two places in a 2APL program: as *guards* in the practical reasoning rules and as *belief query actions* in a plan. We will discuss both of them separately. In what follows, we denote a belief query with β and substitutions with τ.

Practical reasoning rules. As mentioned, 2APL programs may involve three kinds of practical reasoning rules each of which contains a belief query. The three types of practical reasoning rules share the same syntax. A *practical reasoning rule* in 2APL has the form $H \leftarrow \beta \mid \pi$ where H is the head of the rule, β is the guard of the rule representing a belief query, and π is the body of the rule representing a plan. The representation of H is different for each rule type. In case of the PG rule, H is a goal expression represented by a conjunction of positive first-order atoms. For a PC rule, H is either a message, an event or an abstract action represented by a first-order atom. Finally, in case of a PR rule, H is a plan whose execution has failed and is represented by a sequence of actions containing variables. The belief query β may contain conjunctions and disjunctions of first-order literals. A successful query of this guard results in a substitution that can be applied to instantiate variables that occur in the body of the rule. Finally, π is the plan that will be added to the plan base if the rule is applied. The complete description of 2APL constructs can be found in [3].

```
beliefs:
  dist(50).
  new_speed(X) :- X is int(random(10)).

  fuel(1000).
  enough_fuel(X)  :- fuel(Y), X =< Y.

beliefupdates:
  { dist(X) and fuel(F) } Forward(Y)  { not dist(X), not fuel(F), dist(X - Y), fuel(F - Y) }
  { dist(X) and fuel(F) } Backward(Y) { not dist(X), not fuel(F), dist(X + Y), fuel(F - Y) }

goals:
  driveForward(5).

pgrules:
  driveForward(Speed) <- enough_fuel(Speed) and dist(D) | {
    Forward(Speed);
    if B(D <= 0 and new_speed(NewSpeed)) {
      dropgoal(driveForward(Speed));
      adopta(driveBackward(NewSpeed));
    }
  }

  driveBackward(Speed) <- enough_fuel(Speed) and dist(D) | {
    Backward(Speed);
    if B(D >= 100 and new_speed(NewSpeed)) {
      dropgoal(driveBackward(Speed));
      adopta(driveForward(NewSpeed));
    }
  }
```

Fig. 2: Driver: Example 2APL program

An example of a 2APL program is depicted in Figure 2. This program consists of a single agent that will repeatedly move towards and away from a target until it runs out of fuel. The distance `X` of the agent from the target is represented by the belief fact `dist(X)`. Initially, the agent is halfway from the target (the distance is 50) and will start moving forward with a speed of 5 (represented by the goal `driveForward(5)`). This will select the first PG rule, which is applied with the substitution `[Speed/5]` resulting from the unification of the head with the goal base, and the substitution `[D/50]` resulting from the unification of the belief query in the guard of the rule with the belief base. This rule is repeatedly applied until the agent reaches the target (`D <= 0`). Then the goal `driveForward(5)` will be replaced with the goal `driveBackward(NewSpeed)`, where `NewSpeed` is a random integer between 0 and 10. This will activate the second PG rule, which does exactly the opposite as the first PG rule. This process will repeat until the agent runs out of fuel (`enough_fuel(Speed)` can no longer be entailed from the belief base).

As this example might suggest, practical reasoning rules are applied in 2APL in the following way. First the head is instantiated, resulting in a substitution, which we will denote by τ_1. In case of our example, when applying the first PG rule with the head `driveForward(Speed)`, this results in substitution $\tau_1 =$ `[Speed/5]`. Subsequently, the substitution τ_1 is applied to the guard of the rule creating a new belief query which in case of our example is `enough_fuel(5) and distance(D)`. Note that the application of τ_1 to the guard of a rule does not necessarily instantiate all variables involved in the guard (in case of our example variable `D`) such that querying the guard to which τ_1 is applied can result in a new substitution, which we will denote by τ_2. Finally, we would like to emphasize that if there are multiple substitutions for a query possible, then the first substitution is returned. In the case of our example, the new substitution is $\tau_2 =$ `[D/50]`.

Belief Test Action A belief test action occurs in a plan and checks whether the agent has certain beliefs. A belief test action is an expression of the form `B(`ϕ`)`, where ϕ is a belief query represented by a conjunction or disjunction of first-order literals. The execution of a belief test action is basically a belief query to the belief base that can generate a substitution. Since a belief test action occurs in a plan, it may be preceded by some other actions that share variables. This means that some of the variables of a belief test action may already have value instantiation through earlier computed substitutions, which we denote by τ_1 (e.g., substitution resulted from the guard of the practical reasoning rule whose application has generated the plan, or from earlier belief test actions in the same plan). Similar to practical reasoning rules, we first apply the earlier computed substitution τ_1 to the query of the belief test action and then use the new query to check the belief base. The new query will result in a new substitution which we denote by τ_2.

In the case of our example, the belief test action `B(D <= 0 and new_speed (NewSpeed))` contains the variable `D` that is instantiated when the PG rule is applied. This means that τ_1 will contain a substitution for `D`. It also contains the variable `NewSpeed` that is not instantiated before the belief query is performed, which means that it will be instantiated by the belief query. Therefore, τ_2 will contain a substitution for `NewSpeed`.

2.2 Belief updates

2APL contains two different types of belief update actions. The first type of belief update action requires a belief update specification. Each belief update specification is characterized by a triple consisting of the action name represented as an first-order atom starting with a capitalized letter, a precondition represented by a set of first-order literals, and a post-condition that is also represented by a set of first-order literals. One of the belief updates of the example in Figure 2 is:

```
{ dist(X) and fuel(F) } Forward(Y)  { not dist(X), not fuel(F), dist(X - Y), fuel(F - Y) }
```

This triple specifies that any belief update action that unifies with this action name (e.g. `Forward(5)`) can be executed when the pre-condition can be derived from the belief base (when `dist(X) and fuel(F)` can be derived from the belief base for some substitution of the variables `X` and `F`, for instance `distance(50) and fuel(1000)`). The execution of the belief update action ensures that the post-condition is derivable from the belief base (e.g. `not dist(50) and not fuel(1000) and dist(45) and fuel(995)` is derivable from the belief base after the execution of `Forward(5)`). Note that the action call `Forward(5)` will instantiate the variable `Y` and that variable `Y` in the post-condition is instantiated with the same value.

The second type of belief update action does not require a belief update specification and consists of a first-order atom preceded by either the plus (+) or the minus (−) operator. An update action with the plus operator adds the atom to the agent's belief base while an update action with the minus operator will remove the atom from the agent's belief base. For example, the plan "`-dist(50); +dist(45);`" will remove the fact `dist(50)` from the belief base and add the fact `dist(45)` to it. Note that the syntax of simple update actions is the same as the syntax of belief updates in Jason [2].

3 Extending 2APL with Belief Caching

In the previous section, we explained that belief queries demand a substantial amount of processing time of each deliberation cycle and we analyzed belief queries and belief updates in 2APL in order to infer when the result of a belief query will not change and caching can be applied. The answer of a belief query remains unchanged if the following three conditions are satisfied: 1) the part of the belief base that is relevant for the query is not changed, 2) in the case of a practical reasoning rule where the head and the guard share variables, the unification of the head provides a substitution that assigns the same values as the cached values to the shared variables, and 3) in the case of a belief test action that shares variables with some actions that precede it, the substitution originating from the preceding actions assigns the same value as the cached values to the shared variables. As long as these conditions are fulfilled for a belief query β, repeated querying of β returns the same substitutions for its involved variables, such that the query can be cached until one of the conditions is no longer met.

We will illustrate these conditions using the example in Figure 2. Consider the belief query in the guard of the first PG rule (`enough_fuel(Speed) and dist(D)`). The first condition states that the relevant part of the belief base should not be changed for this belief query. This will ensure that two identical belief queries provide the same result. If one of the belief updates `Forward(Y)` or `Backward(Y)` is successfully executed, it will update the value of `dist(X)` in the belief base and thus possibly change the result of the query in the guard of the rule, because this guard contains `dist(D)`. Therefore, the query will have to be performed again and caching does not apply. The second condition states that the substitution of the variables that occur both in head and the guard of the rule should remain unchanged. This means that the substitutions of the variable `Speed` in the rule head should be that same as the previous query, which will ensure that the new belief query in the rule guard `enough_fuel(Speed) and dist(D)` is the same as previous query. The third condition does not apply.

We consider now the belief query action `B(D <= 0 and new_speed(NewSpeed))`. The first condition states again that the belief base should not change in a relevant way. Since no belief update action can update the value of the predicate `new_speed` in the belief base, the result of this query cannot be affected by a belief update action. This means that the first condition is always fulfilled. The second condition does not apply. The third condition states that the variables shared with earlier actions (in this case the instantiation of `D`) should have the same instantiated value as in the previous execution of the query. In our case this means that the earlier substitution resulted from the execution of the belief query `enough_fuel(Speed) and dist(D)` should contain the same value instantiation for the variable `D` as in the current substitution for `D`.

In order to verify whether the first condition holds it is necessary to determine which facts are relevant to belief queries. For this, we calculate the *dependency sets* for all belief queries in a program. The dependency set of a belief query contains all the atoms that can possibly affect the result of the query. Moreover, we calculate the relevant queries for a belief update action as follows: If the post-condition of a belief update action contains an atom that is in the dependency set of a belief query, this query will be added to the list of *relevant queries* for this belief update action. We build our idea of belief caching based on the relevant queries of the update actions. In particular, when the belief update action is invoked, a *changed* flag will be set in its relevant queries. Thus, if the belief base has changed in a relevant way for a belief query, the *changed* flag will be TRUE for this query.

Note that it is possible to calculate the dependency sets of the queries and relevant queries for the belief update actions at compile-time because belief update actions and belief queries are static in 2APL, i.e., no new atoms will be added to the belief base at run-time. This means that this extension will be practically costless in terms of run-time performance. The extension we propose is two-fold. Firstly, the belief queries are extended with a cache to store previous substitutions, a *changed* flag and a decision mechanism to apply caching. Secondly, the definition of a belief update is extended such that it is possible to determine the relevant queries for each belief update. We will explain each extension in more detail in the next two sections.

3.1 Extended Belief Queries

Recall from Section 2.1 that both types of belief queries (as the guard of practical reasoning rules and as belief test actions) involve two substitutions τ_1 and τ_2. τ_1 is the substitution that contains all variables that have been instantiated before the belief query, while τ_2 is the substitution that contains all variables resulting from executing the query to the belief base.

To distinguish between belief queries that contain variables which are already instantiated, i.e. belief queries that contain variables that occur in τ_1, and those that do not, we introduce the flag *shared* for each belief query β and use β*.shared* to refer to this flag. This flag is set (i.e., it has the value TRUE) when the code fragment before the query and the query itself *share* variables. In the case that the query occurs in the guard of a practical reasoning rule, this code fragment is the head of the practical reasoning rule. In the case that the query occurs in a belief query action, the code fragment is the actions that precedes the belief query action.

Definition 1 (Shared belief query). *Let $H \leftarrow \beta \mid \pi$ be a practical reasoning rule and $Var(X)$ is the set of variables that occur in expression X. The flag* shared *of the belief query β is set iff H and β share variables, i.e.:*

$$Var(H) \cap Var(\beta) \neq \emptyset \implies \beta.shared = \text{TRUE},$$

Moreover, let π (the body of the practical reasoning rule) be a plan of the form π';`B(`β`)`;π''. Then, the flag shared *of the belief query β is set iff π' and β share variables, i.e.:*

$$Var(\pi') \cap Var(\beta) \neq \emptyset \implies \beta.shared = \text{TRUE},$$

For example, in Figure 2 the belief queries in both PG rules are shared because the variable `Speed` occurs both in the rule head and rule guard. Similarly, the belief query actions in both rules are shared because the variable `D` occurs both in the rule guard and the belief query action.

In order to perform caching, both substitutions τ_1 and τ_2 are stored for each query β so that they can be re-used for the next query of β. Therefore we introduce for each query β the substitutions τ_1 and τ_2. We cache these substitutions related to query β and denote them by $\beta.\tau_1$ and $\beta.\tau_2$. We would like to emphasize that it may also be possible to store a history of substitutions τ_1 and τ_2 in order to reduce even more queries. This is particularly effective for when τ_1 and τ_2 share variables and τ_1 changes, and the belief base does not change. Next, we introduce for each belief query β the flag *changed* that will be set whenever the belief base has been updated in relevant way, which means that caching does not apply and the query β should be executed with respect to the belief base. The flag *changed* associated with the belief query β is denoted by β*.changed*. This

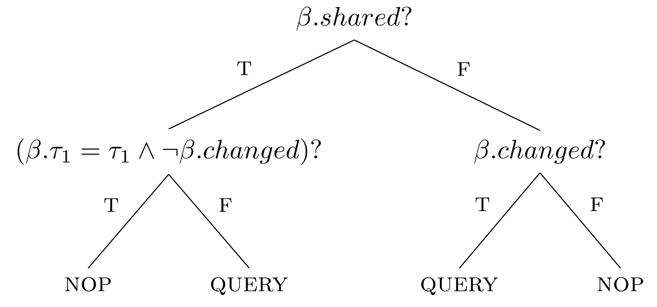


Fig. 3: The belief query caching mechanism.[3]

flag is set by belief update actions, which we will discuss in the next section. For now we simply assume that this flag always has the correct value.

Using these variables it is possible to define a decision mechanism that implements belief caching for the belief queries (Figure 3). If the relevant part of the belief base has been changed for the query β (i.e., $\beta.changed$ is TRUE), the belief query will always be executed. If query β is shared and the cached substitution $\beta.\tau_1$ is different from the current substitution τ_1, the belief query β is executed as well. The reason for this is that the cached substitution $\beta.\tau_1$ applied to β will result in a different query than applying the new substitution τ_1 (which is different from $\beta.\tau_1$) to β. After executing each belief query β, the corresponding flag $\beta.changed$ is set to FALSE.

3.2 Extended Belief Updates

In this section we will define precisely how the caching flag $\beta.changed$ is set for the belief queries. Recall from Section 2 that the only way in which the belief base can be updated is by belief updates. We will make use of *dependency sets* for queries, which we will now introduce. These dependency sets are defined for the belief base of 2APL, which is a general logic program.

Definition 2 (Atom dependency [4]). *An atom a depends on an atom b in a general logic program P iff (i) there exists a clause C in P such that a is the head of C and b occurs in the body of C, or (ii) there exists a clause C in P such that a is the head of C and there is an atom c in the body of C that depends on b.*

Note that the second condition of Definition 2 is recursive, meaning that an atom a can depend on an atom b via any number of clauses $C_1, C_2, ..., C_n$, given that a occurs in the head of C_1, the head of each clause C_i occurs in the body of the previous clause C_{i-1} (given that $i > 1$) and b occurs in the body of C_n.

[3] When a leaf contains NOP, this means that no operation is performed.

Let $\pi(P)$ be the set of atoms occurring in the general logic program P. The atom dependencies in P is a binary relation $R_{dpd} \subseteq \pi(P) \times \pi(P)$.

Definition 3 (Dependency set [4]). *The* dependency set *for an atom a in a general logic program P, denoted by $R^*_{dpd}(a)$, contains all atoms b that a depends on.*

We can calculate the atom dependency set for an atom a using the following two steps, which are a reformulation of the conditions given in Definition 2: 1) Add the atom a in the atom dependency set, 2) Add all atoms occurring in the body of clauses in which atoms in the dependency set occur in the head. Step (2) is repeated until this set does no longer grow. We can straightforwardly extend the definition of an atom dependency set for a belief query.

Definition 4 (Query dependency set). *The* query dependency set *for a query β to a general logic program P, denoted by $R^*_{dpd}(\beta)$, is the union of the atom dependency set of each atom that occurs in β.*

$$R^*_{dpd}(\beta) = \bigcup_{a \in \beta} R^*_{dpd}(a).$$

Suppose a query β is executed at deliberation cycles C_1 and C_2 and that the previous substitution $\beta.\tau_1$ is equal to the current substitution τ_1. The only way in which the result of this query can change is if the substitution in C_2 of a variable X that occurs in an atom in the dependency set of β is different from the substitution of X in C_1. So, if an atom that occurs in the post-condition of a belief update is a member of the query dependency set of a belief query, then that belief update action can affect the substitution of such a variable X.

Consider for instance the belief query `enough_fuel(Speed) and dist(D)` that occurs in the guard of the first PG rule in the example program in Figure 2. According to Definition 4, the query dependency set for a query β is the union of the atom dependency set of each atom that occurs in this query. In this case, this is the union of the atom dependency sets of the atoms `enough_fuel` and `dist`. This is calculuated using the belief base:

```
dist(50).
new_speed(X) :- X is int(random(10)).
fuel(1000).
enough_fuel(X)  :- fuel(Y), X =< Y.
```

We calculate the atom dependency set using the algorithm that we stated directly after Definition 3. First add `enough_fuel` to the set. Then add all atoms occurring in the body of rules in which `enough_fuel` occurs in the head. This means that `fuel` is added to the set, because the last rule in the logic program fulfills this condition. The atom dependency set is now {`enough_fuel`, `fuel`}. After this step, adding atoms that occur in the body of rules in which `enough_fuel` or `fuel` occur in the head does not increase the size of the set, which means that the atom dependency set is complete. Because the atom `dist` does not occur in any clause where there are atoms in the body, the atom dependency set of this atom is simply {`dist`}. This means that the query dependency set of `enough_fuel(Speed) and dist(D)` is {`enough_fuel, fuel, dist`}.

Now, if an atom that occurs in the post-condition of a belief update is a member of this set as well, it can affect the result of this query. Recall that the belief updates of Figure 2 are:

```
{ dist(X) and fuel(F) } Forward(Y)  { not dist(X), not fuel(F), dist(X - Y), fuel(F - Y) }
{ dist(X) and fuel(F) } Backward(Y) { not dist(X), not fuel(F), dist(X + Y), fuel(F - Y) }
```

Since both belief updates contain the atom `dist` and the atom `fuel` and both these atoms occurs in the query dependency set of the belief query `enough_fuel(Speed) and dist(D)`, both belief updates are *relevant* for this query. We make the concept of belief query relevance more clear in the following definition.

Definition 5 (Belief query relevance). *A belief update α is* relevant *for a belief query β if an atom a occurs both in the postcondition of α and in the dependency set of β.*

All relevant queries for a belief update are put in a set and activated whenever the belief update action is executed by setting the *changed* flag of these queries to TRUE.

Definition 6 (Extended belief update). *We add to each belief update α a set relevantQueries containing belief queries and execute the algorithm depicted in Algorithm 1 at compile-time. We also add for each belief update the algorithm depicted in Algorithm 2 that is executed when the belief update action is executed. Call the resulting belief update an* extended belief update*.*

Algorithm 1 Collect relevant queries for each belief update action.

```
1: procedure collectRelevantQueries()
2: for all beliefupdate $\alpha$ do
3:    for all query $\beta$ do
4:       if $\exists p : p \in R^*_{dpd}(\beta) \wedge p \in postcondition(\alpha)$ then
5:          $\alpha.relevantQueries.put(\beta)$
6:       end if
7:    end for
8: end for
9: end procedure
```

3.3 Abstract performance model

The abstract performance model for logic-based agent programming languages, as proposed in [1], can be used in order to measure the effect of belief caching. According to this model, the three steps in the deliberation cycle of a 2APL agent can be mapped onto two kinds of knowledge representation functionality:

Algorithm 2 Reset caching for relevant queries for each belief update action.

```
1: procedure setRelevantQueries ($\alpha$)
2: for query $\beta$ in $\alpha.relevantQueries$ do
3:     $\beta.changed \leftarrow$ TRUE
4: end for
5: end procedure
```

the *query phase* and the *update phase*. Together, they constitute an *update cycle* (Figure 4). The query phase is a phase in which one or more belief queries are performed, and in which no belief updates take place. As soon as a single belief update occurs, the model switches to the update phase. It will remain in the update phase until a single belief query takes place. The belief caching mechanism proposed in [1], which we will call the *original caching mechanism*, is to cache the queries *within one query phase* by making use of a hash table that contains all queries that have been performed in this query phase. This will ensure that the belief base has not been changed, simply because no belief update has occurred. The complete cache is cleared as soon as the model switches to the update phase, i.e. a single belief update takes place.

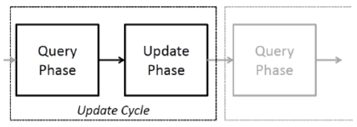


Fig. 4: The abstract performance model [1]

Our implementation is more fine-grained, though, since it refines the general update cycle of [1] to an update cycle *for each individual belief query*. This means that each single belief query goes through the update cycle of Figure 4. Therefore the number of update cycles for individual queries are independent, while in the case of [1] a single belief update will reset the cache of *all* queries. This means that our proposal will lead to more belief caching in the case that the update cycles of the individual belief queries are not identical, a situation which frequently occurs.

4 Experimentation

4.1 Experimental Setup

We have analyzed the working of belief caching using a benchmarking tool that was developed for this work. We have tested belief caching for three increasingly realistic programs.[4]

[4] The sources for the used programs can be downloaded from http://www.students.science.uu.nl/~3714314/2apl_beliefcaching_examples.rar

The first program (`driver`) has been developed to demonstrate the working of belief caching specifically. The code of this program is almost identical to Figure 2, except that the body of the Prolog rule `enough_fuel` has been replaced by a computationally heavy calculation involving integers.

The second program (`storage`) has been written for this task as well but is more realistic. It consists of a multi-agent system with 10 different agents that each can store items in a storage list. Agents will attempt to keep their items stock constant while they receive items from the environment.

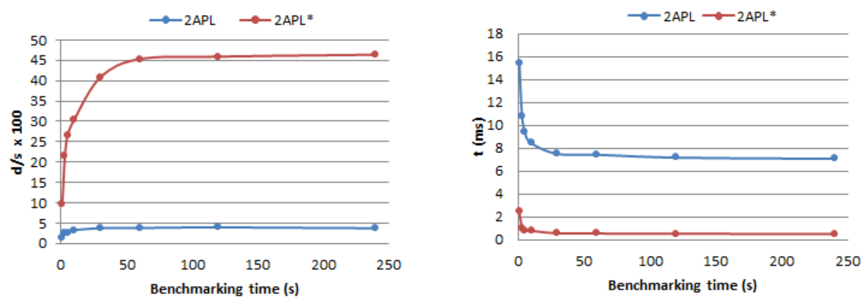
The last program (`marketplace`) is an existing and more sophisticated version of a multi-agent system in which agents have items that they can sell, and have items that they want to buy. Agents can bid for items they desire and sell an item when a bid of another agent meets their demands.

We have compared the results between 2APL with and without belief caching. We use "2APL" to refer to 2APL with no belief caching, and "2APL*" to refer to 2APL with belief caching. All experiments have been performed on a 2.4GHz Intel Core i5, 6 GB 667 MHz DDR3, running Windows 7 and Java 1.6.

When showing the benchmarking results, we use d to denote the number of deliberation steps, Q_b for belief queries, U_b for belief updates. C_{PG} for PG rule calls, C_{PC} for PC rules calls, C_{PR} for PR rule calls, and B for the run-time of the program, which we will also refer to as the benchmarking time.

4.2 Results

Driver program We plot the number of deliberation steps while we increase the run-time of the program (the benchmarking time) (Figure 5a). The asymptotic behavior is due to the fact that initially there exist no cache for the beliefs, but this will be less and less relevant as the benchmarking time increases. The asymptote of 2APL lies around 400 deliberation steps per second, and the one of 2APL* around 4700, which is over ten times as much.



(a) Deliberation steps per second. (b) Average processing time of a PG rule.

Fig. 5: (Driver) Results for increasing benchmarking time.

The only rules that are being used in the `driver` program are PG rules. Therefore, it is of interest to see whether the PG rules are being processed faster because of belief caching. When we plot these values for increasing benchmarking times (Figure 5b), we see that 2APL* processes PG rules much faster than 2APL. Where 2APL has an asymptote at around 7 ms per call, 2APL* has one of around 0.5 ms, which is about 14 times faster.

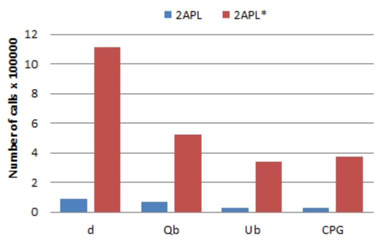


Fig. 6: (Driver) Total number of calls for all operations (B=240s)

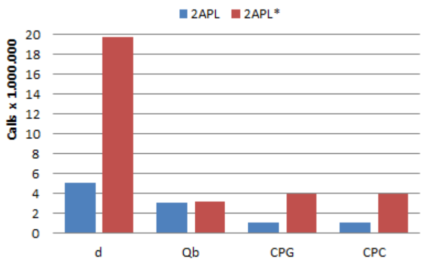
The reason why PG rules are being processed much faster in 2APL* is because less time is spent on doing belief queries. For completeness, the graph showing the total number of calls in 240 seconds for all operations is depicted in Figure 6, which shows that indeed the number of calls for all operations have increased drastically for 2APL*.

Storage program Initial experiments showed that the amsymptote of this program lies around a benchmarking time of 200 seconds. We will therefore limit ourselves to this benchmarking time.

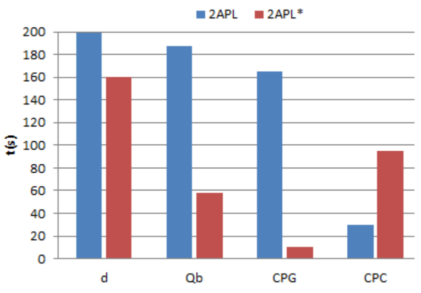
In Figure 7a we have depicted the number of calls for the different operations at a benchmark time of 200 seconds. As we can see, the number of deliberation steps has improved with a factor of about 4 for 2APL*, which is significant. The number of belief queries has remained more of less constant, but since much more deliberation steps have been executed, the number of belief queries per deliberation step has decreased a lot. This is shown more clearly in Figure 7b, where we see that the belief queries take up much less processing time in the case of 2APL*.

`Storage` uses many operations on lists that might not have changed, which means that these operations are omitted when using belief caching. Because the multi-agent system consists of 10 agents, the difference is rather big. We have deliberately used multiple agents to show that difference can become very significant when scaling up the multi-agent system.

Marketplace The last program that we test contains only very simple belief queries. The question that we would like to answer is whether such a program could also be improved using belief caching. As we see in Figure 8a, the number



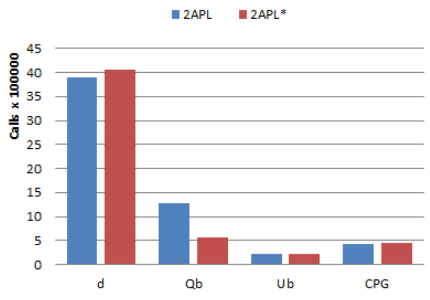
(a) (Storage) Number of calls.



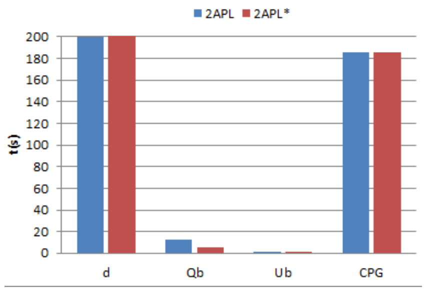
(b) (Storage) Processing time.

Fig. 7: (Storage) Benchmarking results for B=200s.

of deliberation steps increases slightly when using 2APL*, while the number of belief queries decreases with half. This makes sense, because while we save many belief queries, there is not much increase in run-time because the queries are very simple and not time consuming. This becomes more clear in Figure 8b, where the processing time of the different operations is shown. As we can see, the operation time of the belief queries is very small and this does not affect the efficiency of the program greatly.



(a) Number of calls.



(b) Processing time.

Fig. 8: (Marketplace) Benchmarking results for B=200s.

5 Conclusion

We have implemented belief caching into 2APL and showed that it extends the abstract performance model of [1]. Instead of single-cycle caching, our implementation keeps track of an update cycle for each individual belief query. We have implemented belief caching into the latest version of 2APL.

The benchmarking results show that belief caching can optimize a 2APL program significantly, because it is an effective way to reduce the number of belief queries. To what extent this decrease will contribute to an increase in deliberation speed depends on the complexity of the belief queries.

Our most important contribution is that the implementation will never lead to a worse performance, because the dependencies between the belief updates and the belief queries can be calculated at compile-time.

Logic-based agent programming language are based on a combination of imperative programming with logic-based knowledge bases. Because this approach is relatively new, there has not been much research dedicated towards the optimization of the communication between these two formalisms. Our approach has shown that it can be very beneficial to optimize this. We therefore see it as a first step towards increasing the efficiency of logic-based agent programming languages so that they will become better applicable to practical domains.

We plan to continue our optimization work on 2APL by building goal caching mechanism as well as a mechanism that decreases the set of applicable practical reasoning rules. It should be noted that the current 2APL interpreter checks at each deliberation cycle which practical reasoning rule is applicable. This is done by checking the head and guard of the rules which requires queries to belief, goal, and event bases. Any mechanism that keeps track of non-applicable rules may reduce the number of applicable practical reasoning rules and thus the number of time consuming queries.

We believe that our caching mechanism is not limited to 2APL. It can be implemented into logic-based agent programming languages such as Jason [2], GOAL [5], or other multi-agent programming languages that combine logic-based knowledge bases with imperative programming (see [7] for an overview), as long as the set of plan rules do not change at run-time. We leave this issue for further research. Moreover, we believe that the proposed caching mechanism does not change the behavior of 2APL interpreter. In future work, we aim at providing proofs for the soundness and completeness of our caching mechanism.

References

1. Natasha Alechina, Tristan Behrens, Koen Hindriks and Brian Logan, Query Caching in Agent Programming Languages. Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems (ProMAS'12), 117–131, 2012.
2. Rafael Bordini, Michael Wooldridge and Jomi Hübner, Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology). John Wiley & Sons, ISBN 0470029005, 2007.
3. Mehdi Dastani, 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems, 214–248, 2008.
4. Luc De Raedt, Interactive theory revision: an inductive logic programming approach. Academic Press Ltd., ISBN 0-12-210730-6, 1992.
5. Koen Hindriks, Programming Rational Agents in GOAL. Multi-Agent Programming: Languages and Tools and Applications (see [7]), 119–157, 2009.
6. Pouyan Ziafati, Mehdi Dastani, Leon Van Der Torre and John-Jules Meyer, Agent Programming Languages Requirements for Programming Cognitive Robots. Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems (ProMAS'12), 117–131, 2012.
7. Rafael Bordini, Mehdi Dastani, Jurgen Dix and Amal El Fallah Seghrouchni (Eds.), Multi-Agent Programming: Languages, Tools and Applications. Springer, 2009.