Query Caching in Agent Programming Languages

Natasha Alechina[1], Tristan Behrens[2], Koen Hindriks[3], and Brian Logan[1]

[1] School of Computer Science
University of Nottingham
Nottingham NG8 1BB UK
[2] Department of Informatics
Clausthal University of Technology
Germany
[3] Delft University of Technology

Abstract. Agent programs are increasingly widely used for large scale, time critical applications. In developing such applications, the performance of the agent platform is a key concern. Many logic-based BDI-based agent programming languages rely on inferencing over some underlying knowledge representation. While this allows the development of flexible, declarative programs, repeated inferencing triggered by queries to the agent's knowledge representation can result in poor performance. In this paper we present an approach to query caching for agent programming languages. Our approach is motivated by the observation that agents repeatedly perform queries against a database of beliefs and goals to select possible courses of action. Caching the results of previous queries (memoization) is therefore likely to be beneficial. We develop an abstract model of the performance of a logic-based BDI agent programming language. Using our model together with traces from typical agent programs, we quantify the possible performance improvements that can be achieved by memoization. Our results suggest that memoization has the potential to significantly increase the performance of logic-based agent platforms.

1 Introduction

Belief-Desire-Intention (BDI) based agent programming languages facilitate the development of rational agents specified in terms of beliefs, goals and plans. In the BDI paradigm, agents select a course of action that will achieve their goals given their beliefs. To select plans based on their goals and beliefs, many logic-based BDI-based agent programming languages rely on inferencing over some underlying knowledge representation. While this allows the development of flexible, declarative programs, repeated inferencing triggered by queries to the agent's knowledge representation can result in poor performance. When developing multiagent applications for large scale, time critical applications such performance issues are often a key concern, potentially adversely impacting the adoption of BDI-based agent programming languages and platforms as an implementation technology.

In this paper we present an approach to query caching for agent programming languages. Our approach is motivated by the observation that agents repeatedly perform queries against a database of beliefs and goals to select possible courses of action.

Caching the results of previous queries (memoization) is therefore likely to be beneficial. Indeed caching as used in algorithms such as Rete [1] and TREAT [2] has been shown to be beneficial in a wide range of related AI applications, including cognitive agent architectures, e.g., [3], expert systems, e.g., [4], and reasoners, e.g., [5]. However that work has focused on the propagation of simple ground facts through a dependency network. In contrast, the key contribution of this paper is to investigate the potential of caching the results of arbitrary logical queries in improving the performance of agent programming languages. We develop an abstract model of the performance of a logic-based BDI agent programming language, defined in terms of the basic query and update operations that form the interface to the agent's knowledge representation. Using our model together with traces from typical agent programs, we quantify the possible performance improvements that can be achieved by memoization. Our results suggest that memoization has the potential to significantly increase the performance of logic-based agent platforms.

The remainder of the paper is organised as follows. In Section 2 we introduce an abstract model of the interface to a logic-based BDI agent's underlying knowledge representation and an associated performance model. Section 3 presents experimental results obtained from traces of typical agent programs and several key observations regarding query and update patterns in these programs. Section 4 introduces two models to exploit these observations and improve the efficiency of the use of Knowledge Representation Technologies (KRTs) by agent programs. Section 5 discusses related work, and Section 6 concludes the paper.

2 Abstract Performance Model

In this section, we present an abstract model of the performance of a logic-based agent programming language as a framework for our analysis. The model abstracts away details that are specific to particular agent programming languages (such as *Jason* [6], 2APL [7], and GOAL [8]), and focuses on key elements that are common to most, if not all, logic-based agent programming languages.

The interpreter of a logic-based BDI agent programming language repeatedly executes a '*sense-plan-act*' cycle (often called a *deliberation cycle* [9] or *agent reasoning cycle* [6]). The details of the deliberation cycle vary from language to language, but in all cases it includes processing of events (*sense*), deciding on what to do next (*plan*), and executing one or more selected actions (*act*). In a logic-based agent programming language, the *plan* phase of the deliberation cycle is implemented by executing the set of rules comprising the agent's program. The rule conditions consist of queries to be evaluated against the agent's beliefs and goals (e.g., plan triggers in *Jason*, the heads of practical reasoning rules in 2APL) and the rule actions consist of actions or plans (sequences of actions) that may be performed by the agent in a situation where the rule condition holds. In the *act* phase, we can distinguish between two different kinds of actions. *Query actions* involve queries against the agent's beliefs and goals and do not change the agent's state. *Update actions*, on the other hand, are either actions that directly change the agent's beliefs and goals (e.g., 'mental notes' in *Jason*, belief update

actions in 2APL), or external actions that affect the agent's environment, and which may indirectly change the agent's beliefs and goals.

In a logic-based agent programming language, the agent's database of beliefs and goals is maintained using some form of declarative knowledge representation technology. Queries in the conditions of rules and query actions give rise to queries performed against the knowledge representation. Update actions give rise (directly or indirectly) to updates to the beliefs and goals maintained by the knowledge representation. For example, Figure 1 illustrates example rules from *Jason*, 2APL and GOAL agent programs, which select a `move` action to move a block in the Blocks World environment. While the rules appear quite different and have different components, the evaluation of

```
+!on(X,Y) <- !clear(X); !clear(Y); move(X,Y).
```

(a) *Jason*

```
allOnTable <- on(X,Y) and clear(X) and not(Y=table) |
     { @blocksworld(move(X,table),_); On(X,table) }
```

(b) 2APL

```
if a-goal(tower([X| T])) then move(X, table).
```

(c) GOAL

Fig. 1: Example Blocks World rules

the conditions of each rule gives rise to similar queries to the underlying knowledge representation. In this example, the terms `on`, `clear` and `tower` are predicates which are evaluated by querying the belief and goal bases of the agents. Similarly, the agent programs use logical rules (Horn clauses) to represent knowledge about the environment. For example, Figure 2 illustrates a rule used in the *Jason* Blocks World agent to determine whether a set of blocks constitutes a `tower`.

```
tower([X]) :- on(X,table).
tower([X,Y|T]) :- on(X,Y) & tower([Y|T]).
```

Fig. 2: Example *Jason* logical rule

The 2APL and GOAL agents use the same recursive rule in Prolog format with '`&`' replaced by '`,`'. Similarly, in each case, execution of external actions such as `move` and internal belief and goal update actions give rise to updates to the agent's beliefs and goals, either indirectly through perception of the environment (in the case of external action) or directly (in the case of internal actions).

From the point of view of the agent's knowledge representation, the three steps in the *sense-plan-act* cycle can therefore be mapped onto two kinds of knowledge representation functionality. The knowledge representation must provide functionality for

querying an agent's beliefs and goals when applying rules or executing query actions in the agent's plans, and for updating an agent's beliefs and goals upon receiving new information from other agents or the environment, or because of internal events that occur in the agent itself. Our performance model therefore distinguishes two key knowledge representation phases that are common to virtually all logic-based agent programming languages: a *query phase* and an *update phase*. The two phases together constitute an *update cycle*.

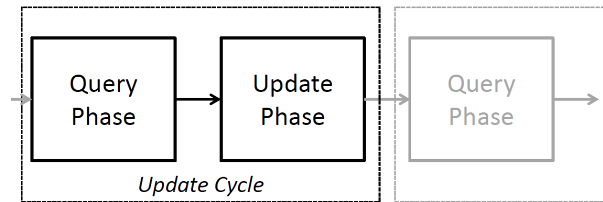

Fig. 3: Update Cycle

The model is illustrated in Figure 3. The query phase includes all queries processed by the agent's knowledge representation in evaluating rule conditions to select a plan or plans, and in executing the next step of the agent's plans (e.g., if the next step of a plan is a belief or goal test action). The update phase includes all updates to the agent's knowledge representation resulting from the execution of the next step of a plan, where this step changes the agent's state directly (e.g., the generation of subgoals or the addition or deletion of beliefs and goals), and updating the agent's state with new beliefs, goals, messages or events at the beginning of the next *sense-plan-act* cycle. Note that update cycles do not necessarily correspond one-to-one to deliberation cycles. For example, in *Jason* and 2APL the action(s) performed at the end of a deliberation cycle may be internal actions (such as test actions) that do not update the agent's beliefs and goals, and in these languages the query phase may include queries from several deliberation cycles. In what follows, we assume that the query phase occurs first and the update phase second, but our results do not depend on this particular order and a similar analysis can be performed if the order of the phases is reversed.

To develop our performance model in detail, we must first make the query/update interface to the agent's knowledge representation precise. Different agent programming frameworks utilise different types of databases to store different parts of the agent's state. For example, most logic-based agent programming languages use different databases for beliefs and for goals, and almost all languages (with the exception of GOAL) maintain bases that store plan-like structures or intentions. Here we focus on those aspects common to most logic-based agent programming languages, namely operations on the agent's beliefs and goals, and abstract away from the details of their realisation in a particular agent platform. In particular, we ignore operations performed on databases of intentions or plans. Although agent platforms do perform operations on

intentions and plans that can be viewed as queries and updates, these operations vary widely from platform to platform and typically do not involve logical inference.

The first key KRT functionality is querying a database. A query assumes the presence of some inference engine to perform the query. In many agent platforms, a distinction is made between performing a query to obtain a single answer and to obtain all answers. In what follows, we abstract away from the details of particular inference engines provided by different agent platforms and represent queries by the (average) time required to perform the query. The second key functionality is that of modifying or *updating* the content of a database. With the exception of recent work on the semantic web and on theory progression in situation calculus, update has not been a major concern for classical (non-situated) reasoners. However it is essential for agents as they need to be able to represent a changing and dynamic environment. All the agent platforms we have investigated use a *simple* form of updating which involves simply adding or removing facts. In cases where the agent platform adopts the open world assumption, one needs to be slightly more general and support the addition and removal of literals (positive and negated facts).

Based on this model, we can derive an analysis of the *average case performance* for a single update cycle of an agent. Our analysis distinguishes between costs associated with the query phase and the update phase of an update cycle. We assume that the agent performs on average N queries in the query phase of an update cycle. If the average cost of a query is c_{qry}, then the average total cost of the query phase is given by

$$N \cdot c_{qry}$$

In general, the same query may be performed several times in a given update cycle. (We provide support for the fact that queries are performed multiple times in a cycle below.) If the average number of unique queries performed in an update cycle is K, then on average each query is performed $n = N/K$ times per cycle.

The total average cost of the update phase of an update cycle can be derived similarly. In logic-based agent programming languages, updates are simple operations which only add or remove facts (literals) from a database, so it is reasonable to consider only the total number of updates when estimating the cost of the update phase. If U is the average number of updates (i.e., adds and deletes) per cycle and c_{upd} is the average cost of an update, then the average total cost of the update phase is given by

$$U \cdot c_{upd}$$

Combining both the query and update phase costs yields:

$$N \cdot c_{qry} + U \cdot c_{upd} \quad (1)$$

3 Experimental Analysis

To quantify typical values for the parameters in our abstract performance model, we performed a number of experiments using different agent platforms and agent and environment implementations. We stress that our aim was not to determine the absolute

or relative performance of each platform, but to estimate the relative average number of queries and updates performed by 'typical' agent programs, and their relative average costs on each platform, in order to determine to what extent caching may be useful as a general strategy for logic-based agent programming languages. To this end, we selected three well known agent platforms (Jason [6], 2APL [7] and GOAL [8]), and five existing agent programs/environments (Blocks World, Elevator Sim, Multi-Agent Programming Contest 2006 & 2011, and Wumpus World).

The agent platforms were chosen as representative of the current state of the art in logic-based agent programming languages, and span a range of implementation approaches. For example, both 2APL and GOAL use Prolog engines provided by third parties for knowledge representation and reasoning. 2APL uses the commercial Prolog engine JIProlog [10] implemented in Java, whereas GOAL uses the Java interface JPL to connect to the open source SWI-Prolog engine (v5.8) which is implemented in C [11]. In contrast, the logical language used in *Jason* is integrated into the platform and is implemented in Java.

The agent programs were chosen as representative of 'typical' agent applications, and span a wide range of task environments (from observable and static to partially observable and real-time), program complexity (measured in lines of code, LoC), and programming styles. The Blocks World is a classic environment in which blocks must be moved from an initial position to a goal state by means of a gripper. The Blocks World is a single agent, discrete, fully observable environment where the agent has full control. Elevator Sim is a dynamic, environment that simulates one or more elevators in a building with a variable number of floors (we used 25 floors) where the goal is to transport a pre-set number of people between floors [12]. Each elevator is controlled by an agent, and the simulator controls people that randomly appear, push call buttons, floor buttons, and enter and leave elevators upon arrival at floors. The environment is partially observable as elevators cannot see which buttons inside other elevators are pressed nor where these other elevators are located. In the 2006 Multi-Agent Programming Contest scenario (MAPC 2006) [13] teams of 5 agents explore grid-like terrain to find gold and transport it to a depot. In the 2011 Multi-Agent Programming Contest scenario (MAPC 2011) [14] teams of 10 agents explore 'Mars' and occupy valuable zones. Both MAPC environments are discrete, partially observable, real-time multi-agent environments, in which agent actions are not guaranteed to have their intended effect.

Finally, the Wumpus World is a discrete, partially observable environment in which a single agent must explore a grid to locate gold while avoiding being eaten by the Wumpus or trapped in a pit. For some of the environments we also varied the size of the problem instance the agent(s) have to deal with. In the Blocks World the number of blocks determines the problem size, and in the Elevator Sim an important parameter that determines the size of a problem instance is the number of people to be moved between floors. The size of problem instances that we have used can be found in the first column of Tables 2 through 6.

It is important to stress that, to avoid any bias due to agent design in our results, the programs were not written specially for the experiments. While our selection was therefore necessarily constrained by the availability of pre-existing code (in particular versions of each program were not available for all platforms), we believe our results are

representative of the query and update performance of a broad range of agent programs 'in the wild'. Table 1 summarises the agents, environments and the agent platforms that were used in the experiments.

Environment	Agent Platform	Agent LoC	Deliberation Cycles
Blocks World	*Jason*	34	104-961
	2APL	64	186-1590
	GOAL	42	16-144
Elevator Sim	2APL	367	3187-4010
	GOAL	87	2292-5844
MAPC 2006	*Jason*	295	2664
MAPC 2011	GOAL	1588	30
Wumpus World	*Jason*	294	292-443

Table 1: Agents and Environments

3.1 Experimental Setup

To perform the experiments, we extended the logging functionality of the three agent platforms, and analysed the resulting query and update patterns in the execution traces for each agent/environment combination. The extended logging functionality captured all queries and updates delegated to the knowledge representation used by the agent platform and the cost of performing each query or update.

In the case of 2APL and GOAL, which use a third party Prolog engine, we recorded the cost of each query or update delegated to the respective Prolog engine. In these languages, Prolog is used to represent and reason with percepts, messages, knowledge, beliefs, and goals. Action preconditions and test goals are also evaluated using Prolog. Prolog queries and updates to the Prolog database therefore account for all costs involved in the query and update phases of an update cycle. In the case of *Jason*, the instrumentation is less straightforward, and involved modifying the JASON belief base to record the time required to query and update percepts, messages, knowledge and beliefs.[4] The time required to process other types of *Jason* events, e.g., related to the intentions or plans of an agent, was not recorded.

We ran each of the agent/environment/platform combinations listed in Table 1 until the pattern of queries and updates stabilised (i.e., disregarding any 'start up' period when the agent(s), e.g., populate their initial representation of the environment). For different agent environments, this required different numbers of deliberation cycles (listed in the *Deliberation Cycles* column in Table 1). For example, fewer deliberation cycles are required in the Blocks World to complete a task than in other environments, whereas in the Elevator Sim environment thousands of deliberation cycles are required to reach steady state. For the real-time Multi-Agent Programming Contest cases, the simulations

[4] In contrast to 2APL and GOAL, *Jason* does not have declarative goals.

were run for 1.5 minutes; 1.5 minutes is sufficient to collect a representative number of cycles while keeping the amount of data that needs to be analysed to manageable proportions. For each agent/environment/platform run, the time required to perform each query or update resulting from the execution of the agent's program was logged, resulting in log files as illustrated in Figure 4. Here, `add` and `del` indicate updates, and

```
...
add on(b2,table) 30
add on(b9,b10) 43
del on(b4,b6) 21
query tower('.'(X,T)) 101
query tower('.'(b7,[])) 51
...
```

Fig. 4: Example log file

`query` indicates a belief or goal query, followed by the updated belief or goal, or query performed, and the time required in microseconds.

3.2 Experimental Results

In this section we briefly present the results of our analysis and highlight some observations relating to the query and update patterns that can be seen in this data. We stress that our aim is not a direct comparison of the performance of the agent programs or platforms analysed. The performance results presented below depend on the technology used for knowledge representation and reasoning as well as on the machine architecture used to obtain the results. As such the figures provide some insight in how these technologies perform in the context of agent programming but cannot be used directly to compare different technologies. Rather our main focus concerns the patterns that can be observed in the queries and updates that are performed by all programs and platforms, and the potential performance improvement that might be gained by caching queries on *each* agent platform.

We focus on the update cycles that are executed during a run of an agent. Recall that these cycles may differ from the deliberation cycle of an agent. An update cycle consists of a phase in which queries are performed which is followed by a subsequent phase in which updates are performed on the databases that an agent maintains. Note that update cycles do not correspond one-to-one to deliberation cycles. In particular, both *Jason* and 2APL agents execute significantly more deliberation cycles than update cycles as can be seen by comparing Table 1 with the tables below. The phases are extracted from log files by grouping query and add/del lines.

We analysed the log files to derive values for all the parameters in the abstract model introduced in Section 2, including the average number queries performed at each update cycle N, the average number of unique queries performed in an update cycle K, the average number of times that the same query is performed in an update cycle N/K, the average cost of a query c_{qry}, the average number of updates performed in an

update cycle U, and the average cost of an update c_{upd}. We also report the number of update cycles for each scenario we have run. Finally, we report the average percentage of queries that are repeated in consecutive update cycles, p. That is, p represents the average percentage of queries that were performed in one cycle and repeated in the next update cycle.

The *Jason* and 2APL agents were run on a 2 GHz Intel Core Duo, 2 GB 667 MHz DDR2 SDRAM running OSX 10.6 and Java 1.6. The GOAL agents were run on a 2.66 GHz Intel Core i7, 4GB 1067 MHz DDR3, running OSX 10.6 and Java 1.6. Query and update costs are given in microseconds. The Size column in Tables 2a – 2c refers to the number of blocks in Blocks world. In Tables 3a and 3b for the Elevator Sim, Size refers to the number of people that randomly are generated by the simulator. The size column in Table 6 refers to the size of grid used: KT2 is a 6×5 grid with one pit, KT4 a 9×7 grid with 2 pits, and KT5 a 4×4 grid with 3 pits.

The results for the Blocks World environment are given in Tables 2a – 2c. Note that the average query and update costs for the GOAL agent decrease when the number of blocks increases. This effect can be explained by the fact that in this toy domain the overhead of translating queries by means of the JPL interface to SWI-Prolog queries is relatively larger in smaller sized instances than in larger sized ones. Also note that the costs found for GOAL agents cannot be used to draw conclusions about the performance of SWI-Prolog because of the significant overhead the Java interface JPL introduces.

Size	N	K	n	p	c_{qry}	U	c_{upd}	Update cycles
10	4.6	3.3	1.39	70%	485	1.1	376	16
50	4.8	3.3	1.46	82%	286	1.0	1057	79
100	5.1	3.3	1.54	82%	317	1.0	2788	152

(a) *Jason*

Size	N	K	n	p	c_{qry}	U	c_{upd}	Update cycles
10	41.1	26.8	1.56	58%	8554	1.8	294	46
50	104.8	78.2	1.34	59%	22335	1.8	273	230
100	235.4	165.8	1.42	59%	49247	1.8	435	460

(b) 2APL

Size	N	K	n	p	c_{qry}	U	c_{upd}	Update cycles
10	26.0	17.6	1.48	59%	89	2.6	58	16
50	100.3	66.0	1.52	63%	64	2.7	35	70
100	153.3	105.7	1.45	70%	59	2.9	29	144

(c) GOAL

Table 2: Blocks World

The results for the Elevator Sim environment are given in Tables 3a and 3b. Tables 4 and 5 give the results for the MAPC 2006 & 2011 environments, and the results for the Wumpus World environment are shown in Table 6.

Size	N	K	n	p	c_{qry}	U	c_{upd}	Update cycles
10	11,214.3	290.3	38.63	52%	97979	2.0	2971	16
50	1,800.7	206.5	8.72	71%	88839	1.1	674	163
100	1,237.7	202.9	6.10	71%	82766	1.1	456	215

(a) 2APL

Size	N	K	n	p	c_{qry}	U	c_{upd}	Update cycles
10	29.5	12.1	1.16	92%	29	1.0	39	5844
50	28.44	23.7	1.20	92%	30	1.0	37	3636
100	34.3	28.6	1.20	90%	31	1.0	36	2292

(b) GOAL

Table 3: Elevator Sim

Size	N	K	n	p	c_{qry}	U	c_{upd}	Update cycles
N/A	4.6	3.7	1.25	76%	256	1.1	96	379

Table 4: Multi-Agent Programming Contest 2006, Jason

Size	N	K	n	p	c_{qry}	U	c_{upd}	Update cycles
N/A	80.5	66.0	1.22	82%	66	28.0	45	30

Table 5: Multi-Agent Programming Contest 2011, GOAL

Size	N	K	n	p	c_{qry}	U	c_{upd}	Update cycles
KT2	8.9	7.4	1.2	59%	671	1.0	173	18
KT4	9.1	7.6	1.2	55%	1170	1.0	166	24
KT5	8.0	6.7	1.2	52%	664	1.0	428	18

Table 6: Wumpus World, *Jason*

As can be seen, even in simple environments like the Blocks World, agent programs may perform many queries in a single update cycle (see Table 2). In the Blocks World experiments, the total number of queries performed during a run ranges from 417 queries for the GOAL agent in the small 10 blocks problem instance in only 16 deliberation cycles to $108,300$ queries for the 2APL agent in the 100 blocks problem instance in 1590 deliberation cycles. Given that the Blocks World environment involves only a small number of beliefs and that the agents use only a few logical rules, this implies that the same query is repeated many times. A similar pattern can be seen in the other experiments. In all cases, the average number of times a query is performed in a single cycle is consistently larger than 1, with N/Kranging from 1.16 (Table 3b) up to 38.63 (3a). Our first observation is therefore that queries are consistently repeated in a single update cycle by all agents in all environments and across all the platforms investigated.

Observation 1 *In a single update cycle, the same query is performed more than once, i.e., we have $n > 1$.*

A second observation that follows consistently from the data is that large percentages of queries are repeated each update cycle. We have found that 22% up to even 92% of queries are repeated in consecutive update cycles.

Observation 2 *A significant number of queries are repeated at subsequent update cycles, i.e., $p > 20\%$.*

Secondly, in all agent/environment/platform combinations investigated, in a single deliberation cycle an agent performs only a few (perhaps only one) actions that directly or indirectly change the state of the agent. This is also supported by the fact that the number of deliberation cycles in most cases is larger than the number of update cycles. In other words, the execution of a deliberation cycle does not always result in an update. Comparing the average number of updates with the average number of unique queries, we consistently find that many more queries are performed than updates in each cycle.

Observation 3 *The number of updates U (add, deletes) performed in an update cycle is significantly smaller than the number of unique queries K performed in that cycle, i.e. $K \gg U$.*

Note that all three observations are independent of the size or complexity of the environment, the complexity of the agent program or the agent platform used. This strongly suggest that the query and update performance of agent programs in the platforms investigated can be significantly improved.

4 Query Caching

The observations in the previous section suggest that efficiency can be significantly increased by memoization, i.e. by caching query results. The cache stores answers to queries, so that if the same query is performed again in the same update cycle, the answers can be returned without recourse to the underlying knowledge representation.

In this section, we first show how to modify the interface to the underlying knowledge representation to incorporate caching. We then extend the abstract performance model introduced in Section 2 in order to analyse the potential increase in performance of caching, and derive a relationship between $n = N/K$ and the costs of maintaining the cache which characterises when caching is likely to be beneficial.

4.1 Extending the Knowledge Representation Interface

The most straightforward approach to exploit the fact that some queries are performed multiple times in a single update cycle, is to add the results of a query to the cache the first time it is performed in an update cycle, and then retrieve the results from the cache if the query is reevaluated at the same update cycle. Although very simple in requiring no information about the average number of times each unique query is repeated in a cycle, as we show below, if the cost of cache insertion is sufficiently low, significant performance improvements can be achieved. Moreover, such an approach requires only a very *loose coupling* between the cache and the underlying knowledge representation.

The cache simply acts as a filter: if a query is a cache hit the results are immediately returned by the cache; if a query is a cache miss, the query is delegated to the knowledge representation and the results stored in the cache, before being returned to the agent program.

The use of a cache requires an extension of the KRT interface with a cache operation `lookup` to lookup entries, an operation `put` to put entries into the cache, and an operation `clear` to clear the cache again. The basic approach can be implemented as shown in the algorithm below.

Listing 1.1: Query Cache

```
1   % Query Phase
2   clear(cache)
3   FOR EACH query Qi DO
4      IF lookup(Qi, answer, cache)
5      THEN return(answer)
6      ELSE DO
7         answer = query(Qi, beliefbase)
8         put(Qi:answer, cache)
9         return(answer)
10     ENDDO
11  ENDDO
```

Of course, by only storing the query results, it is not possible to detect when cache entries are invalidated, so the cache needs to be cleared at the start of each query phase in an update cycle and rebuilt from scratch. In addition, when compiling an agent program, care is required to ensure that differences in variable names are resolved so that similar queries are retrieved from the cache instead of being recomputed. For example, the queries $q(X, Y)$ and $q(A, B)$ which represent the same query but use different variables should not result in a cache miss.

The cache can be implemented by a hash table. Given Observation 2, the size of the hash table can be tuned to optimal size after one or two cycles. By implementing the cache as a hash table, the insertion costs c_{ins} of an entry are constant and the evaluation costs of performing a query a second time are equal to the lookup costs, i.e. a constant c_{hit} that represents the cost for a cache hit. This results in the following performance model, adapted from the model in Section 2:

$$K \cdot (c_{qry} + c_{ins}) + N \cdot c_{hit} + U \cdot c_{upd} \tag{2}$$

It follows that whenever

$$c_{qry} > \frac{K}{N - K} \cdot c_{ins} + \frac{N}{N - K} \cdot c_{hit} \tag{3}$$

it is beneficial to implement a cache. That is, the cache increases performance whenever the average query cost is greater than the average lookup cost of a query plus the average insertion cost times the proportion of unique to non-unique queries (for $N > K$). As expected, the larger the average number of times n a query is performed in a single

cycle, the larger the expected efficiency gains. In the worst case in which all queries are only performed once in a cycle, i.e. $n = 1$, the cache will incur an increase in the cost which is linear in the number of queries, i.e. $N \cdot (c_{ins} + c_{hit})$.

4.2 Experimental Evaluation

To estimate values for c_{ins} and c_{hit} and the potential improvement in performance that may be obtained from caching, we implemented the caching mechanism described in algorithm 1.1 and evaluated its performance by simulating the execution of each agent platform with caching using the query and update logs for the Blocks World and Elevator Sim experiments. The cache is implemented as a single hash table that is filled the first time a query is performed in a query phase and cleared when the first update is performed in the subsequent update phase.

As might be expected, the cost of both cache insertions and hits were low. For our implementation, the cost c_{hit} was about 1 microsecond ($0.45 - 1.16\mu s$) with a similar value for c_{ins} ($0.29 - 0.83\mu s$).

Even in the experiment with the lowest value of n (the Elevator Sim agent programmed in GOAL with 10 people to be transported) the condition of equation 3 is satisfied and performance is improved by caching. In this case, $n = 1.16$ and $c_Q = 29$ (see Table 3b), and we have $29 > 1/0.16 + 1 = 7.25$. The average estimated gain per cycle in this case is $42\mu s$, which can be computed using equation 2 and subtracting the first from the last. The performance gained even in this case is about 10%. In all other cases the gains of using single cycle caching are substantially larger.

5 Related Work

There is almost no work that directly relates to our study of the performance of knowledge representation and reasoning capabilities incorporated into agent programming. As far as we know, our study is the first to investigate patterns in the queries and updates that are performed by agent programs. In [15] it was observed that agent programs appear to spend most of their time in evaluating conditions for adopting plans, although the author's proposed solution was to adopt a plan indexing scheme, rather than to optimize query evaluation in general. In [16] the performance of the FLUX and GOLOG agent programming languages is studied. Another GOLOG-style language, Indi-GOLOG, implements caching [17]. GOLOG-like languages, however, do not implement a deliberation cycle based on the BDI paradigm.

Performance issues of BDI agents have been studied in various other contexts. To mention just a few examples, [18] proposes an extended deliberation cycle for BDI agents that takes advantage of environmental events and [19] proposes the incorporation of learning techniques into BDI agents to improve their performance in dynamic environments. The focus of these papers is on integrating additional techniques into an agent's architecture to improve the performance of an agent instead of on the KRT capabilities of those agents.

6 Conclusion

We presented an abstract performance model of the basic query and update operations that define the interface to a logic-based BDI agent's underlying knowledge representation. Using this model, we analysed the performance of a variety of different agent programs implemented using three different agent platforms. To the best of our knowledge, our study is the first to analyse query and update patterns in existing agent programming languages. Although preliminary, our results suggest that in logic-based agent platforms, knowledge representation and reasoning capabilities account for a large part of the execution time of an agent. In particular, three key observations suggest that integrating memoization into agent programming languages have the potential to significantly increase the performance of logic-based agent platforms: the same queries are performed more than once in a single update cycle, large number of queries are repeated in subsequent cycles, and the number of queries is typically much larger than the number of updates performed.

We showed how the interface to the underlying knowledge representation of an agent platform can be modified to incorporate caching, and extended the abstract performance model to quantify the potential performance improvements that can be achieved by memoization of queries. Our results indicate that even simple query caching techniques have the potential to substantially improve the performance across a wide range of application domains.

The work presented here is limited to a single agent update cycle. Our results, and in particular the observation that a significant number of queries are repeated in subsequent agent cycles, suggests that further performance improvements may be obtained by extending caching to multiple cycles. Extending our abstract performance model and implementation to account for such queries is an area of further work.

References

1. Forgy, C.: Rete: a fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19** (1982) 17–37
2. Miranker, D.P.: TREAT: A better match algorithm for AI production systems. In: Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87), AAAI Press (1987) 42–47
3. Laird, J.E., Newell, A., Rosenbloom, P.S.: SOAR: An architecture for general intelligence. Artificial Intelligence **33** (1987) 1–64
4. Software Technology Branch, Lyndon B. Johnson Space Center Houston: CLIPS Reference Manual: Version 6.21. (2003)
5. Jena. `http://jena.sourceforge.net/` (2011)
6. Bordini, R.H., Hubner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley (2007)
7. Dastani, M.: 2APL: a practical agent programming language. International Journal of Autonomous Agents and Multi-Agent Systems **16** (2008) 214–248
8. Hindriks, K.V.: Programming Rational Agents in GOAL. In: Multi-Agent Programming. Springer (2009) 119–157
9. Dastani, M., de Boer, F.S., Dignum, F., Meyer, J.J.C.: Programming agent deliberation. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, ACM (2003) 97–104

10. JIProlog. http://www.ugosweb.com/jiprolog/ (2011)
11. SWI-Prolog. http://www.swi-prolog.org/ (2011)
12. Elevator Simulator. http://sourceforge.net/projects/elevatorsim/ (2011)
13. Dastani, M., Dix, J., Novak, P.: The first contest on multi-agent systems based on computational logic. In: Proceedings of CLIMA '05. (2006) 373–384
14. Behrens, T., Dix, J., Köster, M., Hübner, J., eds.: Special Issue about Multi-Agent-Contest II. Volume 61 of Annals of Mathematics and Artificial Intelligence. Springer, Netherlands (2011)
15. Dennis, L.: Plan indexing for state-based plans. In: Informal Proceedings of DALT 11. (2011)
16. Thielscher, M.: Pushing the envelope: Programming reasoning agents. In: AAAI Workshop Technical Report WS-02-05: Cognitive Robotics, AAAI Press (2002)
17. Giacomo, G.D., Lespérance, Y., Levesque, H.J., Sardina, S.: IndiGolog: A high-level programming language for embedded reasoning agents. In Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: Multi-Agent Programming: Languages, Platforms and Applications. Springer (2009) 31–72
18. Koch, F., Dignum, F.: Enhanced deliberation in BDI-modelled agents. In: Advances in Practical Applications of Agents and Multiagent Systems (PAAMS 2010), Springer (2010) 59–68
19. Singh, D., Sardina, S., Padgham, L., James, G.: Integrating learning into a BDI agent for environments with changing dynamics. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). (2011) 2525–2530