Multi-Cycle Query Caching in Agent Programming

Natasha Alechina
University of Nottingham
Nottingham, UK
nza@cs.nott.ac.uk

Tristan Behrens
Clausthal University of Technology
Clausthal, Germany
behrens@in.tu-clausthal.de

Mehdi Dastani
Utrecht University
Utrecht, The Netherlands
M.M.Dastani@uu.nl

Koen Hindriks
Delft University of Technology
Delft, The Netherlands
K.V.Hindriks@tudelft.nl

Jomi F. Hübner
Federal University of Santa Catarina
Florianópolis, Brazil
jomi@das.ufsc.br

Brian Logan
University of Nottingham
Nottingham, UK
bsl@cs.nott.ac.uk

Hai Nguyen
University of Nottingham
Nottingham, UK
hhn@cs.nott.ac.uk

Marc van Zee
Utrecht University
Utrecht, The Netherlands
marcvanzee@gmail.com

Abstract

In many logic-based BDI agent programming languages, plan selection involves inferencing over some underlying knowledge representation. While context-sensitive plan selection facilitates the development of flexible, declarative programs, the overhead of evaluating repeated queries to the agent's beliefs and goals can result in poor run time performance. In this paper we present an approach to multi-cycle query caching for logic-based BDI agent programming languages. We extend the abstract performance model presented in (Alechina et al. 2012) to quantify the costs and benefits of caching query results over multiple deliberation cycles. We also present results of experiments with prototype implementations of both single- and multi-cycle caching in three logic-based BDI agent platforms, which demonstrate that significant performance improvements are achievable in practice.

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 (a plan) that will achieve their goals given their beliefs. To select plans, 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, inferencing triggered by repeated queries to the agent's knowledge representation can degrade performance. When developing multi-agent 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 the same queries against a database of beliefs and goals to select possible courses of action (Dennis 2012; Alechina et al. 2012). Caching the results of previous queries (memoization) is therefore likely to be beneficial. Indeed caching as used in algorithms such as Rete (Forgy 1982) and TREAT (Miranker 1987) has been shown to be beneficial in a wide range of related AI applications, including cognitive agent architectures, e.g., (Laird, Newell, and Rosenbloom 1987), expert systems, e.g., (CLIPS 2003), and reasoners, e.g., (Jena 2011). However that work has focused on the propagation of simple ground facts through a dependency network. There has also been considerable work on tabling in Prolog, e.g., (Swift and Warren 2010). However in many cases, tables are cleared after each top-level query. In contrast, the key contribution of this paper is to investigate the potential of caching the results of arbitrary logical queries both within and across deliberation cycles in improving the performance of agent programming languages.

We develop and extend the abstract model of the performance of a logic-based BDI agent programming language presented in (Alechina et al. 2012) to handle multi-cycle caching, defined in terms of the basic query and update operations that form the interface to the agent's knowledge representation. We present empirical results from experiments with prototype implementations of single- and multi-cycle caching in different logic-based BDI agent platforms, which demonstrate that the predicted performance improvements are actually achievable in practice.

Queries and Updates

Agents programmed in a logic-based BDI agent programming language repeatedly execute a '*sense-plan-act*' cycle (often called a *deliberation cycle* (Dastani et al. 2003) or *agent reasoning cycle* (Bordini, Hubner, and Wooldridge 2007)). The details of the deliberation cycle vary from language to language, but in all cases it includes the processing of events (*sense*), deciding on what to do next (*plan*), and executing one or more selected actions (*act*). In a logic-based BDI 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* (Bordini, Hubner, and Wooldridge 2007), the heads and guards of practical reasoning rules in 2APL (Dastani 2008), and mental state conditions in GOAL (Hindriks 2009)), 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* in-

volve 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, and mental state updates in GOAL), or external actions that affect the agent's environment, and which may indirectly change the agent's beliefs and goals.

In such languages, the agent's beliefs and goals are maintained using some form of declarative knowledge representation. From the point of view of the agent's knowledge representation, the three steps in the *sense-plan-act* cycle can be mapped onto two kinds of knowledge representation functionality: *querying* an agent's beliefs and goals when applying rules or executing query actions in the agent's plans; and *updating* an agent's beliefs and goals upon receiving new information from other agents or the environment (in the *sense* phase), or as a result of performing update actions in plans. The answer returned by a query is determined by the agent's current state and hence may be invalidated by an update that changes the state. It is important to note however, that an update can only result in the same query returning a different answer at the *next* deliberation cycle, as updates occur after all queries at the current cycle have been evaluated. If the same query is repeated several times at a given deliberation cycle, caching the results of previous queries (memoization) may therefore be beneficial.

In previous work, Alechina et al. (2012) analysed query and update patterns for a variety of typical agent programs written in the logic-based BDI agent programming languages *Jason* (Bordini, Hubner, and Wooldridge 2007), 2APL (Dastani 2008), and GOAL (Hindriks 2009). Their results showed that for all agent/environment/platform combinations investigated:

- queries are frequently repeated within a single cycle;
- a significant number of queries are repeated at subsequent cycles; and
- 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.

Using data from their experiments, they also developed a simple model of the costs and benefits of caching the answers to queries within a single cycle, and showed, using simulations based on query and update traces from their experiments, that single-cycle caching could be beneficial. However the extent to which the benefits of single-cycle caching predicted by their experiments can be achieved in practice is unclear. Moreover, their analysis did not consider the costs and benefits of multi-cycle caching.

In the remainder of this paper we extend the analysis of Alechina et al. in two ways. First we extend their model to quantify the costs and benefits of *multi-cycle caching*, i.e., of caching the answers to queries over multiple cycles. Second we report the results of experiments with implementations of both single- and multi-cycle caching in three logic-based BDI agent programming languages which demonstrate that the predicted gains of single- and multi-cycle caching are achievable in practice.

Query Caching

In this section, we extend the single-cycle performance model presented in (Alechina et al. 2012) to include multi-cycle caching, and use our extended model to characterise when the benefits of multi-cycle caching outweigh the costs.

Single-Cycle Caching

We begin by summarising the single-cycle model presented in (Alechina et al. 2012). The model is based on the notion of a *query-update cycle* consisting of two phases: a query phase and an update phase. 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 query-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 consecutive deliberation cycles. As in (Alechina et al. 2012) we assume that the query phase occurs first and the update phase second, but the results are the same if the order of the phases is reversed.

If the agent performs on average N queries in the query phase of a query-update cycle, and 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 many cases, the same query is performed several times in the same query phase. If the average number of unique queries performed in a query-update cycle is K, then on average each query is performed $n = N/K$ times per cycle. The average total cost of the update phase of a query-update cycle can be derived similarly. If U is the average number of updates (i.e., adds and deletes) 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:

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

The simplest approach to query caching is to add the results of a query to a cache the first time it is performed in the query phase of a query-update cycle, and then retrieve the results the from the cache if the query is reevaluated in the same query phase. To implement such a cache, the knowledge representation used by the agent must support three operations: `lookup` to lookup entries, `put` to put entries into the cache, and `clear` to clear the cache. The basic approach can be implemented as shown in Algorithm 1 below. As only query results are stored, it is not possible to detect when cache entries are invalidated, so the cache needs to be cleared before the query phase in each query-update cycle by executing the `clear(cache)` command,

and rebuilt from scratch. Although simple, this approach has a number of advantages: it requires no information about the average number of times each unique query is repeated in a query-update cycle; moreover it requires only a very *loose coupling* between the cache and the underlying knowledge representation.

Algorithm 1 Single-Cycle Cache

```
% Query Phase
clear(cache)
for each query Q_i do
    answer ← lookup(Q_i, cache)
    if answer ≠ null then
        return answer
    else
        answer ← query(Q_i, beliefbase)
        put(Q_i, answer, cache)
        return answer
    end if
end for
```

If the cache is implemented as a hash table, the insertion cost c_{ins} of an entry and the lookup cost c_{hit} of a query can be assumed to be constant. This results in the following performance model:

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

It follows that whenever

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

it is beneficial to implement a cache. For a hash table implementation, it is reasonable to assume that $c_{ins} = c_{hit}$, and the above simplifies to

$$c_{qry} > \frac{N+K}{N-K} \cdot c_{hit} \tag{4}$$

That is, the cache increases performance whenever the average query cost is greater than the average lookup cost times the ratio of the total number of queries and unique queries to non-unique queries (for $N > K$). The larger the average number of times n a query is performed in a single query-update 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. $N \cdot (c_{ins} + c_{hit})$.

Multi-Cycle Caching

To exploit the fact that the same queries are performed in subsequent query phases, we need to introduce a cache that persists over multiple query-update cycles. The key challenge here is to check whether cache entries are invalidated in the update phase. In order to implement this check we need an operation `invalidate` that returns those cache entries that are affected by updates. In essence this is a mapping that for each answer for a query in the cache returns the set of facts that are needed to derive that answer. We also need a `delete` operation to remove invalidated cache entries. The approach can be implemented as shown in Algorithm 2 below and extends (Alechina et al. 2012) with changes required for the update phase. Note that there is no need to clear the cache at the beginning of each query phase.

Algorithm 2 Multi-Cycle Cache

```
% Query Phase
for each query Q_i do
    answer ← lookup(Q_i, cache)
    if answer ≠ null then
        return answer
    else
        answer ← query(Q_i, beliefbase)
        put(Q_i, answer, cache)
        return answer
    end if
end for
% Update Phase
queries ← ∅
for each update U_i do
    update(U_i, beliefbase)
    queries ← queries ∪ invalidate(U_i, cache)
end for
for each query Q_j ∈ queries do
    delete(Q_j, cache)
end for
```

Using this model, we can now derive a new performance model for the average cost of a single query-update cycle for the multi-cycle cache. As before, let N represent the average number of queries that are performed in each cycle. By p we denote the percentage of the N queries that are repeated at the next cycle and not invalidated by updates. Thus we have $p \cdot N$ queries that have been performed before and are still in the cache. The remaining $(1-p) \cdot N$ queries are the new queries in each cycle. We assume that the K unique queries at each cycle are uniformly distributed between the $p \cdot N$ cached and $(1-p) \cdot N$ uncached queries. This seems reasonable: N/K is largely determined by the agent's program, while p is largely determined by the agent's environment. N and K are determined by the plan triggers in the agent's program, and by queries in it's currently executing plan(s). (This holds whether the queries in a plan trigger are evaluated as a single boolean expression or 'left to right'.) On the assumption that queries in executing plans are less numerous than plan trigger queries (there are typically many plan triggers and only a few, or only one, executing plans), then the queries performed at the current cycle and their results will only differ from the queries performed at the previous cycle if the agent's beliefs and goals have changed. In general, changes in the agent's beliefs and goals are predominantly a result of perception of its environment or interaction with other agents, rather than plan execution, and there is no a priori reason why the parts of the agent's program and the queries evaluated as a result of such changes (previously unevaluated queries if query evaluation is left to right, or new answers to a a previously evaluated query) would have a different ratio of N/K. There is no reason to suppose, for ex-

ample, that N/K would be lower for queries which are new at this cycle.

The performance model is based on the pseudocode of Algorithm 2 and consists of two terms. The first term represents the costs of the query phase:

$$N \cdot c_{hit} + (1-p) \cdot K \cdot (c_{qry} + c_{ins}) \tag{5}$$

As above c_{ins} denotes the cache insertion cost and c_{hit} denotes the cost of looking up a cache entry. The second term represents the costs of the update phase:

$$U \cdot (c_{upd} + c_{chk}) + (1-p) \cdot K \cdot c_{del} \tag{6}$$

where as above U denotes the average number of updates performed in the update phase. We take the number of cache removals that need to be performed as a result of cache entries being invalidated by updates to be $(1-p) \cdot K$. The cost of checking whether an update invalidates cache entries is represented by c_{chk} and the cost of deleting a cache entry is represented by c_{del}. The average total cost then results from adding the query and update costs.

If we denote the total number of cached queries by $R = N - ((1-p) \cdot K)$ and assume that $c_{ins} = c_{hit}$ (as above) , it follows that whenever

$$c_{qry} > \frac{N + (1-p) \cdot K}{R} \cdot c_{hit} + \frac{U}{R} \cdot c_{chk} + \frac{(1-p) \cdot K}{R} \cdot c_{del} \tag{7}$$

it is beneficial to implement a multi-cycle cache. It is instructive to compare how the multi-cycle cache compares with the single-cycle cache. The multi-cycle cache outperforms the single- cycle cache whenever:

$$p \cdot K \cdot c_{qry} > U \cdot c_{chk} + (1-p) \cdot K \cdot c_{del} \tag{8}$$

i.e., whenever the query costs associated with repeating $p \cdot K$ unique queries are higher than the costs associated with checking whether updates invalidate cache entries and deleting those entries. Since the costs of deleting a cache entry are negligible compared to checking whether a cache entry is invalidated, the key factor is the cost of checking for invalidated cache entries. Given that $K \gg U$, this means that the multi-cycle cache outperforms the single cycle cache whenever the percentage p of repeated queries is sufficiently high. As it is relatively easy to monitor p, it is also possible to switch between single-cycle and multi-cycle caching at runtime.

Experimental Evaluation

To determine whether the potential performance gains of single- and multi-cycle caching are achievable in practice, we performed experiments with single- and multi-cycle caching in a number of different agent programming platforms, and evaluated the average cost of queries and updates with and without caching for a range of different agent programs. The agent platforms used, *Jason* (Bordini, Hubner, and Wooldridge 2007), 2APL (Dastani 2008), and GOAL (Hindriks 2009), differ significantly in architecture and implementation, and the caching implementations were done independently by developers familiar with the platform. The consistent improvements in performance we report below are therefore likely to hold for caching implementations in other logic-based BDI agent languages.

We implemented the single-cycle caching mechanism (Algorithm 1) in each of the three platforms. Due to time limitations, multi-cycle caching (Algorithm 2) was implemented only for 2APL and GOAL. To determine the performance of the caching implementations, we performed experiments with five existing agent programs/environments (Blocks World, Grid World, Elevator Sim, Multi-Agent Programming Contest, and Wumpus World), and compared the average time to perform a query and an update both with and without caching for each platform. We also report the average percentage of queries that resulted in cache hits.[1] 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), and programming styles. 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 a version of each program was 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'.

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. The Grid World is a single agent, discrete, partially observable environment in which bombs must be removed by bringing them to a trash bin that is located somewhere on the grid. 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. 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. 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) (Dastani, Dix, and Novak 2006) teams of 5 agents explore a grid-like terrain to find gold and transport it to a depot. In the 2011 and 2012 Multi-Agent Programming Contest scenario (MAPC 2011, MAPC 2012) (Behrens et al. 2011) 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 ef-

[1] Although for comparison of agent platforms benchmarks are arguably necessary, our aim here is not to determine the absolute or relative performance of each platform, but to determine the benefits of caching for each platform.

fect. 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, in the Grid World the number of bombs determines the size of the problem, and in the Elevator Sim we varied the number of people to be moved between floors.

To perform the experiments, we extended the logging functionality of each agent platform to capture all queries and updates delegated to the platform's knowledge representation, and the CPU time (in microseconds) required to perform each query and update. In the case of 2APL and GOAL, which use third party Prolog engines, we recorded the cost of each query or update delegated to the respective Prolog engine. In the case of *Jason*, the instrumentation was less straightforward, and involved modifying the *Jason* belief base to record the CPU time required to query and update percepts, messages, knowledge and beliefs. The time required to process other types of *Jason* events, e.g., related to the intentions or plans of an agent, was not recorded. All platforms implement update using basic add and delete operations on facts, and the belief queries as well as the belief updates that occur in the agent programs are static in the sense that they are not modified at run-time.

The logged values were used to compute the average cost of queries c_{qry} and updates c_{upd} for each agent/environment/platform combination. To focus on the parameters in the performance model, we have normalised the average times by taking the lowest value of c_{qry} and c_{upd} for each platform as the unit value (1) for that platform: all other values are reported as multiples of the unit value. This has the advantage of abstracting from the particulars of machines and underlying knowledge representation used by different platforms and gives a more uniform presentation of the results. We also report the percentage of the total number of queries that resulted in cache hits (denoted by h in the tables below).

The *Jason* agents were run on a 2 GHz Intel Core Duo, 2 GB 667 MHz DDR2 SDRAM and the GOAL agents on a 2.66 GHz Intel Core i7, 4GB 1067 MHz DDR3, both running OSX 10.6 and Java 1.6. The 2APL agents were run on a 2.4GHz Intel Core i5, 6 GB 667 MHz DDR3, running Windows 7 and Java 1.6. We report on the results obtained for each platform below.

Jason The single-cycle caching mechanism given in Algorithm 1 was implemented in *Jason* with one minor difference: the cache is used only for queries whose answer(s) depend on rules (clauses). For queries that are simple beliefs, the cache is skipped since the *Jason* belief base already stores such basic beliefs directly in a hash table. As is to be expected, the number of cache hits is therefore lower, and this is apparent when comparing the h values reported in Table 1 with those for Table 2 and Table 3. In general, the percentages for *Jason* are significantly lower than for either 2APL or GOAL.

Even so, in general, the results for *Jason* show that using a cache reduces the average query time. Gains of up to 20% were found, for example, for the Blocks World domain. In this domain, queries are relatively expensive compared to those performed in the MAPC environments. The speedup is therefore more significant in the Blocks World domain than in the MAPC domain. Given that multi-cycle caching generally increases the number of cache hits (cf. Tables 2 and 3), we would expect significant performance gains of multi-cycle caching for *Jason*.

Problem	Caching	h	c_{qry}	c_{upd}
Blocksworld 10	No	0%	132.40	225.70
Blocksworld 10	Single-cycle	18%	124.58	225.70
Blocksworld 50	No	0%	180.45	527.37
Blocksworld 50	Single-cycle	37%	170.39	527.37
Blocksworld 100	No	0%	172.63	441.90
Blocksworld 100	Single-cycle	41%	143.58	441.90
MAPC 2006	No	0%	1.21	41.90
MAPC 2006	Single-cycle	7%	1.00	41.90
MAPC 2012	No	0%	3.45	541.34
MAPC 2012	Single-cycle	7%	3.45	541.34

Table 1: Comparison of different caching models - *Jason*

2APL In the case of 2APL, multi-cycle caching has been incorporated in the Java implementation of the 2APL interpreter to allow most of the computation required for the `invalidate` operation to be performed at compile-time, rather than at run-time.[2] The caching implementation exploits the fact that belief queries and updates are static in a 2APL program. First, the set of belief queries that may be affected by a belief update action are associated with that belief update. Because the rules (clauses) in the belief base are static as well, we can compute at compile-time which belief queries may be affected by a belief update action. This set of queries is computed by constructing a query dependency graph that is derived from the rules. Using this graph the atoms that a query depends on can be determined. When a belief update action is executed in the update phase, all belief queries that depend on the updated atom(s) are removed from the cache. At each query phase, all belief queries that are not present in the cache are queried on the belief base, while the others are retrieved from the cache. Single-cycle caching is implemented by clearing the cache at the beginning of each query-update cycle.

The results of experiments with single- and multi-cycle caching in 2APL are shown in Table 2. In all cases it is clear that using caching decreases execution time. While it does not seem to matter much whether we use single-cycle caching or multi-cycle caching in the Grid World application, in the Wumpus World this is not the case. The number of cache hits increases significantly with multi-cycle caching, and the execution time decreases as well. This suggests that both types of caching can improve efficiency of 2APL applications significantly.

GOAL For GOAL, both single- and multi-cycle caching were implemented. The implementation of single-cycle caching is straightforward, and follows closely the pseudocode in Algorithm 1. In what follows, we focus on the implementation of multi-cycle caching.

[2]Belief caching is part of the latest 2APL release available at *http://apapl.sourceforge.net.*

Problem	Caching	h	c_{qry}	c_{upd}
Gridworld 10	No	0%	3.61	2.61
Gridworld 10	Single-cycle	51%	2.82	5.75
Gridworld 10	Multi-cycle	52%	2.93	1.92
Gridworld 50	No	0%	1.81	1.58
Gridworld 50	Single-cycle	51%	1.51	2.97
Gridworld 50	Multi-cycle	52%	1.35	3.12
Gridworld 100	No	0%	1.41	1.63
Gridworld 100	Single-cycle	51%	1.21	1.46
Gridworld 100	Multi-cycle	52%	1.00	1.75
Wumpus Small	No	0%	34.64	2.13
Wumpus Small	Single-cycle	54%	35.99	3.55
Wumpus Small	Multi-cycle	80%	14.13	3.91
Wumpus Large	No	0%	82.41	2.64
Wumpus Large	Single-cycle	57%	52.56	2.72
Wumpus Large	Multi-cycle	84%	41.62	3.04

Table 2: Comparison of different caching models - 2APL

The multi-cycle caching implementation extends the single-cycle implementation as in Algorithm 2. To implement the `invalidate` operation, we used a meta-interpreter written in Prolog that, in addition to the answer to a query, returns the ground facts used to answer the query. Calls to SWI-Prolog are replaced by calls to the meta-interpreter. The answer to each query is stored in a hash table *queryCache*. Each ground fact f returned by the meta-interpreter is also stored together with the set of queries it may invalidate, *invalidates(f)* in a hash table. In later query-update cycles, if an update (insertion or deletion) of f is performed then, for each query in *invalidates(f)*, its cached result is removed from *queryCache*, and f is also removed from *invalidates*. Note that, in contrast to the 2APL implementation, computation of dependency information is performed at run-time rather than compile-time.

Although experiments show that the average query times for calls to the meta-interpreter are about 1.5 to 2 times higher than normal queries, as we clear the cache less often, the number of calls to SWI-Prolog decreases resulting in a reduction in average query times compared to single-cycle caching. Table 3 shows the comparison between different caching models in GOAL.

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, Alechina et al. (2012) are the first to investigate patterns in the queries and updates performed by agent programs. We extend this work by presenting a multi-cycle model and empirical analysis of this model. Dennis (2012) has observed that agent programs appear to spend most of their time in evaluating conditions for adopting plans; however the solution proposed was to adopt a plan indexing scheme, rather than to optimise query evaluation in general. Thielscher (2002) has studied the performance of the FLUX and GOLOG agent programming languages, and another GOLOG-style language, Indi-GOLOG, implements caching (De Giacomo et al. 2009). However GOLOG-like languages do not implement a deliberation cycle based on the BDI paradigm.

Problem	Caching	h	c_{qry}	c_{upd}
Blocksworld 10	No	0%	18.76	18.35
Blocksworld 10	Single-cycle	27%	15.37	16.17
Blocksworld 10	Multi-cycle	36%	15.24	14.10
Blocksworld 50	No	0%	14.65	15.64
Blocksworld 50	Single-cycle	32%	13.54	15.22
Blocksworld 50	Multi-cycle	51%	11.08	13.01
Blocksworld 100	No	0%	12.92	14.51
Blocksworld 100	Single-cycle	31%	11.51	14.98
Blocksworld 100	Multi-cycle	54%	10.53	13.21
Elevator 10	No	0%	6.67	6.92
Elevator 10	Single-cycle	83%	1.18	7.15
Elevator 10	Multi-cycle	90%	1.00	7.02
Elevator 50	No	0%	6.90	6.69
Elevator 50	Single-cycle	65%	2.57	6.97
Elevator 50	Multi-cycle	79%	2.02	6.21
Elevator 100	No	0%	7.05	6.66
Elevator 100	Single-cycle	65%	2.65	6.94
Elevator 100	Multi-cycle	77%	2.13	6.34

Table 3: Comparison of different caching models - GOAL

Performance issues of BDI agents have been studied in various other contexts. To mention just a few examples: Koch and Dignum (2010) propose an extended deliberation cycle for BDI agents that takes advantage of environmental events and Singh et al. (2011) propose 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 rather than single- or multi-cycle query caching.

Conclusion

We extended the abstract performance model of the query and update operations that define the interface to a logic-based BDI agent's underlying knowledge representation presented by Alechina et al. (2012) to quantify the costs and benefits of multi-cycle caching. To the best of our knowledge, our study is the first to present and analyse a model for caching over multiple deliberation cycles in agent programming languages. We also showed how the interface to the underlying knowledge representation of an agent platform can be modified to incorporate multi-cycle caching, and analysed the performance of both single and multi-cycle caching using a variety of different agent programs implemented using three different logic-based BDI agent programming languages. Although based on a relatively small number of example programs, our experimental results indicate that both a single and a multi-cycle caching techniques have the potential to substantially improve the performance of logic-based BDI agent programming languages across a range of application domains.

Our focus has been on the knowledge representation and reasoning capabilities of agents. In logic-based agent platforms these capabilities account for a large part of the performance of an agent. Future work is needed to build a performance model that accounts for all aspects in an agent's deliberation cycle and to put our work into perspective relative to this bigger picture.

References

Alechina, N.; Behrens, T.; Hindriks, K.; and Logan, B. 2012. Query caching in agent programming languages. In Dastani, M.; Logan, B.; and Hübner, J. F., eds., *Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems (ProMAS 2012)*, 117–131.

Behrens, T. M.; Dix, J.; Hübner, J.; and Köster, M. 2011. Special issue: The multi-agent programming contest: Environment interface and contestants in 2010, editorial. *Annals of Mathematics and Artificial Intelligence* 61(4):257–260.

Bordini, R. H.; Hubner, J. F.; and Wooldridge, M. 2007. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley.

Software Technology Branch, Lyndon B. Johnson Space Center, Houston. 2003. *CLIPS Reference Manual: Version 6.21*.

Dastani, M.; de Boer, F.; Dignum, F.; and Meyer, J.-J. 2003. Programming agent deliberation: an approach illustrated using the 3APL language. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, 97–104. New York, NY, USA: ACM Press.

Dastani, M.; Dix, J.; and Novak, P. 2006. The first contest on multi-agent systems based on computational logic. In Toni, F., and Torroni, P., eds., *Computational Logic in Multi-Agent Systems, 6th International Workshop, CLIMA VI, London, UK, June 27-29, 2005, Revised Selected and Invited Papers*, 373–384. Springer.

Dastani, M. 2008. 2APL: a practical agent programming language. *Journal of Autonomous Agents and Multi-Agent Systems* 16(3):214–248.

De Giacomo, G.; Lespérance, Y.; Levesque, H. J.; and Sardina, S. 2009. IndiGolog: A high-level programming language for embedded reasoning agents. In Bordini, R. H.; Dastani, M.; Dix, J.; and Fallah-Seghrouchni, A. E., eds., *Multi-Agent Programming: Languages, Platforms and Applications*. Springer. 31–72.

Dennis, L. 2012. Plan indexing for state-based plans. In Sakama, C.; Sardiña, S.; Vasconcelos, W.; and Winikoff, M., eds., *Declarative Agent Languages and Technologies IX - 9th International Workshop, DALT 2011, Taipei, Taiwan, May 3, 2011, Revised Selected and Invited Papers*, volume 7169 of *LNCS*, 3–15. Springer.

Forgy, C. 1982. Rete: a fast algorithm for the many pattern/-many object pattern match problem. *Artificial Intelligence* 19(1):17–37.

Hindriks, K. V. 2009. Programming rational agents in goal. In El Fallah Seghrouchni, A.; Dix, J.; Dastani, M.; and Bordini, R. H., eds., *Multi-Agent Programming: Languages, Tools and Applications*. Springer US. 119–157.

Jena. 2011. `http://jena.sourceforge.net/`.

Koch, F., and Dignum, F. 2010. Enhanced deliberation in BDI-modelled agents. In Demazeau, Y.; Dignum, F.; Corchado, J.; and Bajo-Perez, J., eds., *Advances in Practical Applications of Agents and Multiagent Systems (PAAMS 2010)*, volume 70 of *Advances in Soft Computing*, 59–68. Springer.

Laird, J. E.; Newell, A.; and Rosenbloom, P. S. 1987. SOAR: An architecture for general intelligence. *Artificial Intelligence* 33:1–64.

Miranker, D. P. 1987. TREAT: A better match algorithm for AI production systems. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87)*, 42–47. AAAI Press.

Singh, D.; Sardina, S.; Padgham, L.; and James, G. 2011. Integrating Learning into a BDI Agent for Environments with Changing Dynamics. In Walsh, T., ed., *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 2525–2530.

Swift, T., and Warren, D. S. 2010. XSB: Extending Prolog with tabled logic programming. *CoRR* abs/1012.5123.

Thielscher, M. 2002. Pushing the envelope: Programming reasoning agents. In *AAAI Workshop Technical Report WS-02-05: Cognitive Robotics*. AAAI Press.