Lazy Evaluation and Subsumption Caching for Search-Based Integrated Task and Motion Planning

Christian Dornhege Andreas Hertle Bernhard Nebel

***Abstract*— State of the art classical planning systems can efficiently solve large symbolic problem instances. Applying classical planning techniques to robotics is possible by integrating geometric reasoning in the planning process. The problems that are solvable in this way are significantly smaller than purely logical formulations as many costly geometric calculations are requested by a planner. Therefore we aim to avoid those calculations while preserving correctness.**

We address this problem with efficient caching techniques. *Subsumption caching* avoids costly computations by caching geometric queries and beyond answering the same queries also considers less or more constrained ones. Additionally, we describe a lazy evaluation technique that pushes applicability checks for successor states performing geometric queries to a later point. As we are interested in the performance of our planner not as a standalone component, but as part of an intelligent robotic system, we evaluate those techniques embedded in an integrated system during real-world mobile manipulation experiments.

I. INTRODUCTION

Solving complex tasks that combine many different skills is an essential problem for mobile manipulation robots in a household scenario. Applying the robot's capabilities to the task at hand requires a behavior that utilizes the available skills in a goal-directed manner. Skills such as picking up an object or moving the robot can be viewed as symbolic actions. Finding an action sequence that leads to the desired goal is no trivial task [1] as one needs to consider constraints between heterogeneous actions for arbitrary situations and the possible combinations easily grow very large. This is a task that a symbolic planner is well suited for. However, one cannot abstract away the geometric constraints that arise when acting in the real-world. For example consider the cleaning task in Figure 1: The robot must not only ensure that the object standing on the spot to clean is removed, but that a collision-free trajectory exists for the wipe motion.

Therefore many state of the art systems use an integrated task and motion planner [2], [3] that incorporates arbitrary reasoners in the symbolic planning process. In addition many systems [4], [5], including ours, don't follow a plan blindly once it has been computed, but use a continual planning approach. The planner is embedded in an observation, monitoring and execution loop. During the execution of a plan the robot's state is estimated and monitored by the planner to verify that the current plan still leads to the goal. If this is not the case, replanning is triggered to come up with a plan that fits the current situation. This helps to recover from execution failures and to react to unexpected percepts in a partially observable environment. Still, guaranteeing successful completion of complex tasks is a hard problem that requires some assumptions to be made about the world [5]. We also address those assumptions allowing our robot to act under real-world constraints such as partial observability and non-deterministic actions [6]. A more detailed description of our full system is available in our previous work [7].

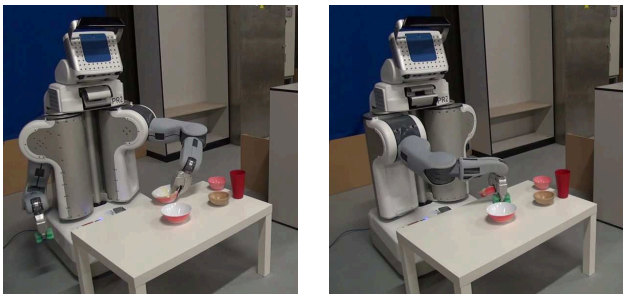

Fig. 1. This figure shows the PR2 robot during the execution of a tidy up task. The robot picks up a bowl to wipe the spot where it was placed.

In this paper we focus on the efficiency of integrated task and motion planning specifically for search-based planners. We integrate arbitrary reasoners in our planner Temporal Fast Downward/Modules (TFD/M) using the concept of semantic attachments [8]. During the search process external modules query these reasoners for geometric facts like "Where can I put this cup on the table?" Such operations take several orders of magnitude longer than determining the truth value of purely symbolic facts. Not surprisingly a substantial part of the planning time is consumed by external computations. Therefore our main goal is to avoid such module calls whenever possible without losing the soundness of the produced plans.

We define several caching techniques that store computed results and are applicable to any integrated reasoning system. Partial state caching uses a minimal cache key to represent a query, so that the same computation is never done twice. Our first contribution is extending this method to subsumption caching that is able to answer queries if they are subsumed by another cached query. Second, we introduce a lazy evaluation method that postpones module calls arising during successor generation and thereby avoids geometric computations for states in the search queue. Third, we evaluate these techniques in real-world experiments on

All authors are members of the University of Freiburg, Department of Computer Science, 79110 Freiburg, Germany, *email:{dornhege,hertle,nebel}@informatik.uni-freiburg.de*.

This work was partially supported by DFG grants SFB/TR-8 project R7, EXC 1086 BrainLinks-BrainTools, and grant NE 623/13-1.

the PR2 robot solving mobile manipulation tasks. Besides comparing the effectiveness of our methods, we present an overview of the overall system performance using an integrated task and motion planner in a realistic application scenario.

II. RELATED WORK

Applying high-level reasoning techniques to real-world robotics problems requires the integration of logic-based and geometric representations. Beetz et al. [9] developed a plan based controller. Learned action sequences are stored in a plan library to be combined and adapted to the current situation and task. Kresse and Beetz [10] use constraints as a common symbolic and geometric interpretation to efficiently solve complex problems guided by a symbolic plan.

Producing plans that are sound on the symbolic as well as the geometric level requires an integration of task and motion planning. One way is to use a symbolic planner to guide expansions of roadmaps by imposing constraints on geometric configurations [1]. Our previous work focused on integrating arbitrary reasoners directly into the planning process [8]. Current approaches can deal with incomplete knowledge and beliefs while integrating geometric reasoning into the planning process [2], [4].

Computing motion plans as part of a symbolic plan is a costly process. Srivastava et al. [11] interleave task and motion planning and feed symbolic facts determined by motion planners back into the classical planner to avoid committing early to instantiations of continuous operators. Another way is to *plan in the now* [5] using abstract versions of operators for plan steps far enough in the future. Other approaches focus on reuse of solutions: in their combined task and motion planner Wolfe et al. [3] use subtask-specific irrelevance to reuse trajectories that have been computed before. This concept is similar to what we call partial state caching, which we extend even further to subsumption caching. In the context of classical planning Eyerich et al. investigated subsumptions of planning operators [12], although without connecting this concept to geometric constraints.

In classical planning heuristics can be approximated by the heuristic value of the parent state, so that computationally expensive heuristics are never computed for states in the search queue [13]. We use a similar concept to avoid geometric computations for queued states.

III. INTEGRATED TASK AND MOTION PLANNING WITH SEMANTIC ATTACHMENTS

We will now state basic definitions that we use to describe a planning task and then discuss, how generic motion planners are integrated by using semantic attachments.

A. Planning Task

Our planner TFD/M uses the well known planning domain definition language (PDDL) to describe planning tasks. It is originally based on Fast Downward [14] that grounds and translates PDDL definitions to a finite domain representation (FDR). Our definitions are derived from the work by Helmert [15], but adapted to focus on the parts relevant for integrating task and motion planning. In particular we deal only with Boolean and numerical values and ignore conditional effects.

Definition 1. *A **planning task** is a tuple $P = (V, V_N, I, G, O)$, where V is a finite set of logical variables, V_N is a finite set of numerical variables, I is the initial state, G is a goal formula in propositional logic and O a finite set of operators.*

The set of all possible variable assignments defines the set of all possible states—the state space S.

Definition 2. *A (full) **state** s is a function $s : V \cup V_N \to \{true, false\} \cup \mathbb{R}$, where $s(v) \in \mathbb{R}$ if $v \in V_N$ and $s(v) \in \{true, false\}$ if $v \in V$. For a propositional formula ϕ, $\phi(s)$ denotes the evaluation of ϕ on the variable assignment s.*

Transitions from one state to another are defined by operators.

Definition 3. *An **operator** o is a tuple $(\phi, e, cost)$, where ϕ is a propositional formula about the set of variables V called the **precondition**, e an **effect**, and $cost \in \mathbb{R}+$. We write $cost(o)$ to denote the cost of an operator o.*

*An operator o is **applicable** in state s or $applicable(s, o)$ is true, iff $\phi(s)$ is true.*

Effects are applied to a state s in the usual way following PDDL semantics. A plan for a planning task P is a finite sequence of applicable operators that when applied to the initial state lead to the goal.

B. Semantic Attachments

For now we only described classical planning tasks without any integration of other reasoners. Also, the set of numerical variables V_N is not utilized, although numerical values are surely relevant to describe real-world problems. PDDL [16] provides a numerical extension, but its expressiveness is not general enough to handle robotics tasks.

The idea behind semantic attachments is that certain parts of a planning task—namely variable evaluations in applicability checks, effects on numerical variables, and operator costs—can have semantics attached to them that are computed by an external module.

A *condition checker* module allows to determine truth values of logical variables by an external function call. We use *condition checkers* to determine if an object can be placed on a table. Consider the following excerpt from the `putdown-object` action:

```
(:action putdown-object
  :parameters (?l - location ?o - movable
    ?s - static ?a - arm)
  :precondition
    (and (grasped ?o ?a)
      ([canPutdown ?o ?a ?s ?l]) ...)
```

Square braces signal that `canPutdown` has a semantic attachment. Our implementation for `canPutdown` uses a

discretization of the table surface and determines if there are any reachable poses given the attached object and configuration of objects on the table. For wiping a *condition checker* tests whether a wipe spot is free of obstructing objects.

An *effect applicator* allows to update numerical state variables based on the computation of an external function. Similarly to the `canPutdown` module, we define the effect applicator `updatePutdownPose` in the `putdown-object` action. The declaration of the `updatePutdownPose` module is given by:

```
(updatePutdownPose ?o - movable
  ?a - arm ?s - static ?l - location
  (x ?o) (y ?o) (z ?o)
  (qx ?o) (qy ?o) (qz ?o) (qw ?o)
  effect updatePutdownPose@libput.so)
```

The list `(x ?o) ... (qw ?o)` gives the list of numerical variables to be updated—in this case the new 6-dof pose of the object to be placed that we derive from the same computation as the `canPutdown` module.

A *cost module* computes the operator cost by an external function. We provide a *cost module* for navigation that calls a path planner [17] and returns the path cost. A set of navigation locations is given to the robot along with a 3d map [18] to the navigation module.

C. Search

We define the planning problem as finding a valid plan with minimal cost for a given task P. Our planner uses forward-chaining best first search in the state space. An open queue is seeded with the initial state. In each step the first state is removed from the open queue, which is sorted by heuristic estimates of the states. If this state fulfills the goal formula, a plan was found. Next, all operators are tested for applicability in this state and the successors are in turn added in the open queue.

As finding minimal cost plans is often hard in practice one aims to find a *satisficing* plan that might not be optimal, but has satisfying costs. For this reason the heuristic needs not to be admissible. Often the search continues even after a plan is found until either the open queue is empty or a timeout is reached.

IV. CACHING TECHNIQUES

In relation to classical planning applying an operator or just testing for applicability can take orders of magnitude longer when external modules are called. Therefore the main goal of the techniques introduced in this section is to avoid as many module calls as possible while retaining soundness.

One way is to cache results to module calls. Different types of modules might require the same computation. Therefore, computation results are stored instead of just the return values for a specific module and the same cache is exchanged between compatible modules. For example, our `canPutdown` *condition checker* stores the computed putdown pose of the object. Subsequent calls to the `updatePutdownPose` *effect applicator* can now access the pose without recomputing the motion. We name a call to any type of module a *request*. The caching techniques presented in this section are able to store computation results for such requests and thus can be applied generally in any system that issues such requests.

A. Full State Caching

The full state s paired with the operator o that the request was issued from is used as a key. The cache hits those cases, where the same request might be issued multiple times, e.g. when revisiting a state that a shorter path was found to.

B. Partial State Caching

A partial state $s^p \sqsubseteq s$ is computed by the module and used as the cache key together with the operator o.

Definition 4. *A partial state $s^p \sqsubseteq s$ is a partial variable assignment, i.e. a function $s^p : V^p \rightarrow \{true, false\}$ and $s^p : V_N^p \rightarrow \mathbb{R}$, where $V^p \subseteq V$ and $V_N^p \subseteq V_N$.*

This partial state must contain all relevant information for the computation. A minimal partial state is straight-forward to determine as it consists of all variables used in the module computation. Consider a putdown motion that only cares about objects near the robot. A partial state implicitly defines the set of states $S^p = \{s' \in S | s^p \sqsubseteq s'\}$. Any request for a state in S^p will hit the same cache entry. $|S^p|$ determines the advantage of partial state over full state caching.

C. Subsumption Caching

We take the idea of a partial state one step further and include requests for different states. The idea is to build a subsumption hierarchy[1] of more or less constrained states.

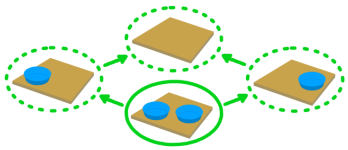

Fig. 2. This figure shows the subsumption hierarchy for successful requests to put an additional object on the table. When a putdown request succeeded on the table with two objects already placed, we can infer that less constrained cases must also succeed without the need to do any calculations.

Take the example of a putdown request for an object that succeeded when two objects were already placed on the table. Now if only one object were present, a request will surely succeed as it is less constrained (always given the same object positions and dimensions). This is illustrated in Figure 2. On the other hand, when a putdown request failed with three objects present, requests with four objects must also fail. We use this principle in our putdown modules.

Definition 5. *For a state s and operator o a state s^- is less constrained, i.e., $s^- \leq_o s$ iff $applicable(s, o) \Rightarrow applicable(s^-, o)$. Likewise, a state s^+ is more constrained iff $s \leq_o s^+$.*

[1]This term is not to be confused with the well known subsumption architecture. Here we mean logical subsumption.

Algorithm 1 Subsumption Caching request for s, o

Cache Mapping $Successes : S \times O \to Result$
Cache Mapping $Failures : S \times O \to Result$

for all $(s', o' \mapsto r') \in Successes$ **do**
  if $s \leq_o s'$ **and** $o = o'$ **then**
    return r'
for all $(s', o' \mapsto r') \in Failures$ **do**
  if $s \geq_o s'$ **and** $o = o'$ **then**
    return r'
// Cache miss, compute and insert into cache
$r, success \leftarrow computeRequest(s, o)$
if $success$ **then**
  for all $(s', o' \mapsto r') \in Successes$ **do**
    if $s \geq_o s'$ **and** $o = o'$ **then**
      // Subsumed by s, o
      Remove $(s', o' \mapsto r')$ from $Successes$
  Insert $(s, o \mapsto r)$ into $Successes$
else
  for all $(s', o' \mapsto r') \in Failures$ **do**
    if $s \leq_o s'$ **and** $o = o'$ **then**
      Remove $(s', o' \mapsto r')$ from $Failures$
  Insert $(s, o \mapsto r)$ into $Failures$
return r

For a (partial) state s and operator o, we test if s is less constrained than some state s', where the request succeeded. Analogously we check whether s is more constrained than some state s'', where the request failed. Should neither test match, the request is computed and the result added to the cache. We build a subsumption hierarchy of cached requests by only retaining the most constrained states for succeeded requests and the least constrained states for failed requests in the cache. Algorithm 1 illustrates the procedure. Note that in comparison to the first two caching methods the constraint relation is domain dependent. Without a non-trivial constraint relation subsumption caching becomes partial state caching. Thus, if such a relation can be found, subsumption caching dominates partial state caching.

D. Global caching

As we use continual planning, the planner might be called multiple times during a task. Global caching reuses cache entries from previous planner calls. To ensure soundness, we require that the cached information fully defines the module computation. This technique is complementary to the above mentioned and thus can be combined with either.

V. Lazy Module Evaluation

Lazy module evaluation tackles the same point in the search as a technique from classical planning named *deferred heuristic evaluation* [13]. Many successor states are generated and pushed in the open queue, but might never be visited. Deferred heuristic evaluation identifies heuristic calculations as the expensive operation. By taking the heuristic measure of the successor's parent instead of the successor itself, only a single heuristic computation is needed for all successors. In our case, generating successors is the costly part as this means computing $applicable(s, o)$, which might call modules. Therefore, we defer these calculations to a later time by using a relaxed version of an operator to perform the applicability-check.

Definition 6. *The operator o^+ is a relaxation of an operator o if for all states $s \in S$: $applicable(s, o) \Rightarrow applicable(s, o^+)$.*

This will never discard applicable states, but might include inapplicable ones. Therefore, we cannot compute a successor state and instead store the pair s, o implicitly defining the successor. The additional problem that we cannot calculate a heuristic value without the successor is easily solved by applying deferred heuristic evaluation. The issue of possibly inapplicable open queue entries is addressed once a state is taken out of the queue to be expanded. If $applicable(s, o)$ fails at this point, we continue by taking the next state out of the open queue. Note that in comparison to replacing applicability checks by faster, approximate versions this method is sound.

We derive relaxed operators if the operator condition is a conjunction containing module variables as positive literals, which is often the case. The simplest option is to skip all module calls. A better way is offered by our module interface that optionally calls a fast, relaxed version. Often there is an obvious way to come up with such a relaxation. Consider for example the `canPutdown` condition checker: Instead of computing a motion plan, the relaxation could test whether the goal configuration is collision-free.

VI. Evaluation

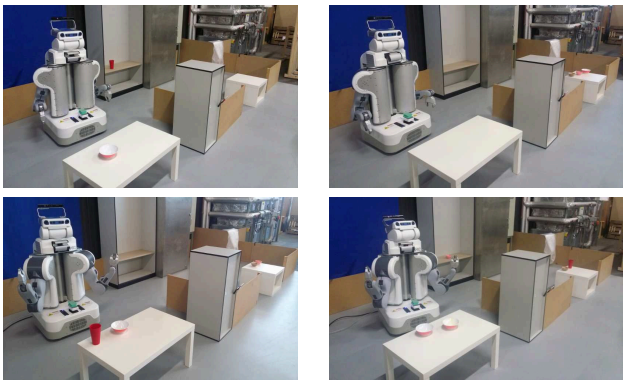

Fig. 3. This figure shows four of the evaluation settings. The task was to bring all objects to the front table and wipe the spot under each object.

We evaluated our system on the PR2 robot. One to five objects were positioned in two different configurations for each number of objects (see Figure 3). The goal was to find all objects, bring them to a specific table and wipe the initial location of each object, so each object had to be interacted with at least once. Pick, place and object detection functionality was provided by the standard software for the PR2 [19]. Wiping followed a minimal traveling salesman path in joint space [20].

Each setting was executed until the system reported the task to be completed. For each call we ran the planner with lazy and eager module evaluation. After the first plan was found the anytime search continued for another 25% of the time. This way we automatically adapt to the problem complexity. The plan from the lazy solution was executed. We used partial state caching and stored results globally.

A. System Performance

We investigate the overall system performance in Table I. For each task we give the total accumulated planning time for lazy and eager module evaluation, as well as the maximum time for a single plan. Single plan times show how hard the task itself was once all objects had been seen for a first time. Subsequent planner calls might have cached computations or face a problem that is already partially solved by the robot's previous actions. During the experiments two tasks were not fully completed. In task 6 one wipe action could not be executed and in task 9 the robot placed an object too far on the edge dropping it unrecoverably. Nevertheless, the system continued to solve all remaining goals.

As expected we observe increasing planning and execution times the more objects are present. The relation between execution and total planning time indicates the usability as a practical system. Besides the number of objects additional factors are the initial object placements and the order in which the robot finds and interacts with objects. If the robot solves the task partially for some objects before finding others, later planning calls might be easier. Such eagerness might be advantageous, but can have adversary effects if those solved objects block others later on. For smaller problems planning time was considerably lower than for larger ones. We see that in relation to the execution time planning scales slightly worse, which is not a surprise given the combinatorial nature of the problem. Comparing lazy and eager module evaluation we observe that lazy evaluation is able to find plans faster than eager evaluation.

B. Caching Techniques

We perform a detailed comparison of the different caching techniques. For each module call, we determined if each caching strategy produced a hit or miss. We also recorded the time it would have taken to compute the query for a miss. Recorded times for all queries are accumulated over each run. Module computations were only performed if partial state caching had a miss. No caching or full state caching misses would repeat identical computations and in these cases we use the stored time from partial state caching. If subsumption caching misses, we use the time from partial state caching. Note that we must perform all partial state computations, even if subsumption caching hits as this might have been subsumed from a different partial state. We omit the time it takes to answer a cache hit, which in our case is much lower than the computation time for a miss.

Table II shows caching misses and times for putdown module calls. As expected, no caching or full state caching are quite inefficient and not feasible in practice. Also there is no difference for full state caching with or without global caching, mainly due to the fact that the full state contains the current robot state, which is unlikely to be exactly the same between planner calls. Partial state caching performs considerably better than those techniques as it never repeats the same calculation. Subsumption caching is able to reduce cache misses even further. We presume that the impact would be greater for even more complex problems.

Global caching is also able to reduce cache misses, mainly in tasks 6 and 7. Table I shows that those tasks required multiple re-planning steps that were able to utilize the stored computations. Although subsumption caching dominates partial state caching, in task 6 we see more misses for subsumption caching than partial caching in conjunction with global caching. Those were due to numerical inaccuracies when converting global cache keys back to poses, which is only required when matching states in subsumption caching. We also investigated the navigation module and partial caching proved similarly effective. As subsumption caching is not applicable in this case, we omit those results for space reasons.

VII. CONCLUSIONS

We presented caching and lazy evaluation techniques implemented in our integrated task and motion planner and investigated their efficiency in real-world experiments when the planner was embedded in a high-level continual planning executive. The system was able to successfully handle unexpected events and adapt to new situations by re-planning. Lazy module evaluation notably reduces planning time for search-based planners.

The caching techniques are not limited to search based planning. Full state caching is easy to implement, but not as effective as partial state caching, which however needs a case-by-case implementation. Global caching is complementary to other caching strategies and has proven useful for longer tasks with numerous re-plannings.

Subsumption caching works, but its efficiency is dependent on the ability to create a subsumption hierarchy and thus shows mostly in complex tasks. We believe that the idea of subsuming requests can be exploited further. A generic implementation is possible if modules supply which state variables were relevant to a certain query, ideally providing minimal reasons. For example, a putdown request that fails because no inverse kinematics solution was found is independent of any other objects.

The presented techniques were able to reduce planning times to a good balance between planning and execution time. There is also room for improvement on the planning side. One approach lies in the fact that classical planners are not tailored to integrated task and motion planning. In this work, we looked at avoiding module calls. Given they make up a major part of planning time, spending more time in elaborate, more informed heuristics that take relaxed versions of modules into account is worth investigating. Improved search guidance will focus on calls relevant to the goal. The techniques presented in this paper are complementary to such

Task	Objects	Total Planning Time [s]		Max Single Plan Time [s]		Planner Calls	Execution Time [s]	Actions
		Eager	Lazy	Eager	Lazy			
1	1	76.7	41.8	72.0	37.3	2	497.2	24
2	1	66.0	43.0	61.2	39.0	2	303.4	21
3	2	57.4	42.3	47.9	32.2	3	807.8	38
4	2	106.2	71.2	73.2	42.2	4	631.8	40
5	3	221.0	158.1	112.6	76.6	4	823.8	46
6	3	124.0	99.8	42.4	27.7	18	1630.9	94
7	4	289.6	220.8	203.1	153.5	10	1226.1	63
8	4	120.9	99.8	57.8	56.4	4	1019.3	55
9	5	686.2	505.3	263.7	220.1	6	1651.7	82
10	5	350.2	255.5	281.3	211.0	3	1195.0	56

TABLE I

OVERVIEW OF THE SYSTEM PERFORMANCE UNDER EAGER AND LAZY EVALUATION.

Task	Number of Requests	No Caching Time [s]	Full State		Full State (G)		Partial State		Partial State (G)		Subsumption		Subsumption (G)	
			Misses	Time [s]	Misses	Time [s]	Misses	Time [s]	Misses	Time [s]	Misses	Time [s]	Misses	Time [s]
1	98	60.5	60	36.7	60	36.7	8	4.5	8	4.5	8	4.5	8	4.5
2	140	101.0	70	49.7	70	49.7	5	3.0	5	3.0	5	3.0	5	3.0
3	347	170.4	224	110.5	224	110.5	22	11.3	17	8.9	20	10.5	15	8.0
4	1124	898.7	402	318.9	402	318.9	37	27.0	32	23.0	37	27.0	32	23.0
5	7488	4701.7	1954	1184.1	1954	1184.1	70	46.4	62	42.2	68	45.3	62	42.2
6	2559	1422.1	1568	862.4	1568	862.4	200	121.7	68	46.8	178	111.7	95	71.1
7	5234	3411.2	1880	1285.5	1848	1259.7	175	140.2	90	69.1	159	126.6	74	55.5
8	1167	790.9	540	358.3	540	358.3	66	40.3	66	40.3	66	40.3	66	40.3
9	11823	15907.6	3980	5301.0	3980	5301.0	247	472.1	228	413.1	203	393.7	190	340.9
10	6900	4230.3	1844	1112.8	1844	1112.8	63	38.8	63	38.8	62	37.6	62	37.6

TABLE II

COMPARISON OF THE DIFFERENT CACHING METHODS FOR THE PUTDOWN MODULE. METHODS ANNOTATED WITH (G) ALSO USED GLOBAL CACHING.

improvements and thus are an efficient part of any integrated task and motion planning system.

REFERENCES

[1] S. Cambon, R. Alami, and F. Gravot, "A hybrid approach to intricate motion, manipulation and task planning," *Int. Journal of Robotics Research*, vol. 28, no. 1, pp. 104–126, 2009.

[2] L. Kaelbling and T. Lozano-Perez, "Integrated task and motion planning in belief space," *Int. Journal of Robotics Research*, 2013.

[3] J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation," in *Int. Conference on Automated Planning and Scheduling (ICAPS)*, 2010.

[4] A. Gaschler, R. P. A. Petrick, T. Kröger, A. Knoll, and O. Khatib, "Robot task planning with contingencies for run-time sensing," in *Workshop on Combining Task and Motion Planning at ICRA*, 2013.

[5] L. Kaelbling and T. Lozano-Perez, "Hierarchical task and motion planning in the now," in *IEEE Conference on Robotics and Automation (ICRA)*, 2011.

[6] B. Nebel, C. Dornhege, and A. Hertle, "How much does a household robot need to know in order to tidy up your home?" in *AAAI Workshop on Intelligent Robotic Systems*, 2013.

[7] C. Dornhege and A. Hertle, "Integrated symbolic planning in the tidyup-robot project," in *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI II*, 2013.

[8] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel, "Semantic attachments for domain-independent planning systems," in *Int. Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, September 2009, pp. 114–121.

[9] M. Beetz, T. Arbuckle, M. Bennewitz, W. Burgard, A. Cremers, D. Fox, H. Grosskreutz, D. Haehnel, and D. Schulz, "Integrated plan-based control of autonomous service robots in human environments," *IEEE Intelligent Systems*, vol. 16, 2001.

[10] I. Kresse and M. Beetz, "Movement-aware action control – integrating symbolic and control-theoretic action execution," in *IEEE Conference on Robotics and Automation (ICRA)*, 2012.

[11] S. Srivastava, L. Riano, S. Russell, and P. Abbeel, "Using classical planners for tasks with continuous operators in robotics," in *ICAPS Workshop on Planning and Robotics (PlanRob)*, 2013.

[12] P. Eyerich, M. Brenner, and B. Nebel, "On the complexity of planning operator subsumption," in *Int. Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, 2008, pp. 518–527.

[13] S. Richter and M. Helmert, "Preferred operators and deferred evaluation in satisficing planning," in *Int. Conference on Automated Planning and Scheduling (ICAPS)*, 2009, pp. 273–280.

[14] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[15] ——, "Concise finite-domain representations for PDDL planning tasks," *Artificial Intelligence*, vol. 173, pp. 503–535, 2009.

[16] M. Fox and D. Long, "PDDL2.1: an extension to PDDL for expressing temporal planning domains," *Journal of Artificial Intelligence Research (JAIR)*, vol. 20, no. 1, pp. 61–124, 2003.

[17] M. Likhachev, G. J. Gordon, and S. Thrun, "ARA*: Anytime A* with provable bounds on sub-optimality," in *Advances in Neural Information Processing Systems 16*. MIT Press, 2004.

[18] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013.

[19] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[20] J. Hess, D. Tipaldi, and W. Burgard, "Null space optimization for effective coverage of 3d surfaces using redundant manipulators," in *Int. Conference on Intelligent Robots and Systems (IROS)*, 2012.