2014 IEEE International Conference on Robotics & Automation (ICRA)
Hong Kong Convention and Exhibition Center
May 31 - June 7, 2014. Hong Kong, China

Cache-Aware Asymptotically-Optimal Sampling-Based Motion Planning

Jeffrey Ichnowski, Jan F. Prins, and Ron Alterovitz[1]

***Abstract*— We present CARRT* (Cache-Aware Rapidly Exploring Random Tree*), an asymptotically optimal sampling-based motion planner that significantly reduces motion planning computation time by effectively utilizing the cache memory hierarchy of modern central processing units (CPUs). CARRT* can account for the CPU's cache size in a manner that keeps its working dataset in the cache. The motion planner progressively subdivides the robot's configuration space into smaller regions as the number of configuration samples rises. By focusing configuration exploration in a region for periods of time, nearest neighbor searching is accelerated since the working dataset is small enough to fit in the cache. CARRT* also rewires the motion planning graph in a manner that complements the cache-aware subdivision strategy to more quickly refine the motion planning graph toward optimality. We demonstrate the performance benefit of our cache-aware motion planning approach for scenarios involving a point robot as well as the Rethink Robotics Baxter robot.**

I. Introduction

Incremental sampling-based motion planners are a critical component of many robotic systems that autonomously navigate and/or manipulate objects [1]. The objective of motion planning is to compute a feasible path from a starting configuration to a goal while avoiding obstacles. Asymptotically-optimal incremental sampling-based motion planners, such as the Rapidly-exploring Random Tree (Star) (RRT*), converge towards a plan that minimizes a cost function by incrementally refining the planning graph data structure [2]. In this paper, we introduce CARRT*, "Cache-Aware Rapidly-exploring Random Tree (Star)", an asymptotically-optimal sampling-based motion planner that significantly reduces motion planning computation time by effectively utilizing the cache memory hierarchy of modern central processing units (CPUs).

Modern CPUs can perform hundreds of computation instructions in the time that it takes to access a single value in memory (RAM) [3]. To reduce this disparity, CPUs have multiple levels of small and fast cache memories for storing frequently accessed data and avoiding costly trips to RAM. Fig. 2 shows a typical modern CPU with 3 levels of cache: its L1 cache is the smallest and fastest (30–50$\times$ faster than RAM), L2 is bigger and not as fast (12–20$\times$ faster than RAM), and L3 is largest but slow (2–5$\times$ faster than RAM).

CARRT* is an asymptotically-optimal sampling-based motion planner that is *cache-aware*—it takes into account

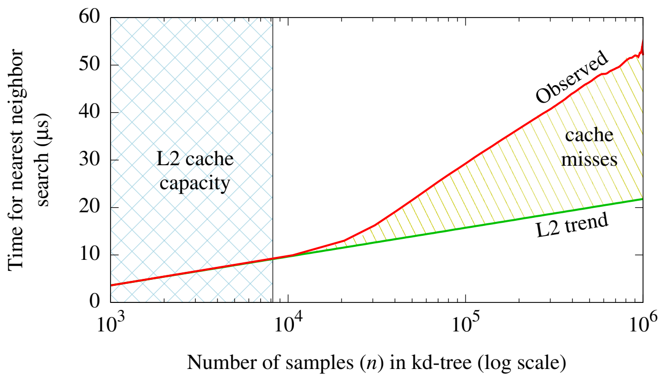


Fig. 1. Nearest neighbor searching is a critical component of sampling-based motion planning. Proper use of the CPU's cache can lead to significantly faster nearest neighbor searches. As the number of configurations in the space rises, the memory required to store the nearest neighbor search data structure (e.g., a kd-tree) exceeds the capacity of the CPU's L2 cache. This results in L2 cache misses, and the associated latency causes the observed nearest neighbor search times to diverge from the trend seen when the data fits in L2 cache. In this paper, we present a motion planner that is *cache-aware*—with a simple tunable parameter, it keeps its working dataset in the CPU cache. This results in computation times closer to the L2 cache trend line (in green) than the observed red line, enabling significant improvements in motion planning performance.

the size of the cache to organize its computations in a manner that significantly increases the number of cache hits. We focus on two portions of the algorithm that have increasing memory complexity as the algorithm iterates: nearest neighbor searching and graph rewiring.

Nearest neighbor searching is a critical component of sampling-based motion planning, and the computational complexity grows logarithmically with the number of sampled configurations in the motion planning graph. As the number of sampled configurations rises, the nearest neighbor search data structure exceeds the capacity of the CPU's cache levels. The result is *cache misses* where the cache does not contain a requested value. As shown in Fig. 1, the impact of cache misses is significant; nearest-neighbor search times diverge from the trend seen when the data structure fits completely in L2 cache.

Rather than exploring anywhere in configuration space in every iteration as in RRT*, CARRT* focuses on exploring in distinct smaller regions of the configuration space for short periods of time. As CARRT* adds more configurations, it progressively subdivides regions to keep the working dataset under a preconfigured limit. By tuning the region size limit to match the characteristics of the problem and the CPU cache size, CARRT* works with a dataset that fits in the cache. Computation times thus become closer to what would

[1]Jeffrey Ichnowski, Jan F. Prins, and Ron Alterovitz are with the Department of Computer Science, University of North Carolina at Chapel Hill, USA `{jeffi,prins,ron}@cs.unc.edu`
This research was supported in part by the National Science Foundation (NSF) through awards IIS-1117127 and IIS-1149965 and by the National Institutes of Health (NIH) under award R21EB017952.

be possible if RAM operated as fast as the cache, enabling significant improvements in motion planning performance.

RRT* and CARRT* incrementally converge towards optimality by *rewiring* the planning tree around configurations as they add them. Because CARRT* samples in regions, it would take longer for rewiring to have a global impact were it to follow the same rewiring approach of RRT*. We thus develop a rewiring strategy compatible with cache-aware region-based sampling and that accelerates computation of high quality motion plans.

We evaluate CARRT* in scenarios involving a point robot as well as the Rethink Robotics Baxter robot [4]. Our results show that the cache-aware approach of CARRT* outperforms non-cache-aware RRT*.

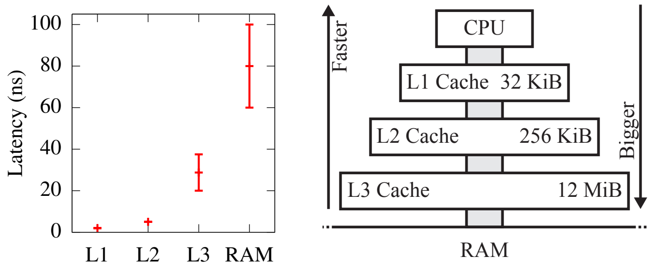
II. Related Work

CARRT* uses a cache-aware region-based sampling strategy. Non-uniform sampling in a sampling-based planner has been a subject of considerable research. Hsu et al. provide an overview of many sampling strategies in their approach that adaptively chooses among several samplers [5].

Sampling within a bounded region of the configuration space has been used to varying effects. RESAMPL [6] uses sampling to classify regions and then refine sampling within the regions based upon their classification to help solve difficult planning problems such as narrow passages. PRRT* [7] uses a simple partitioning scheme to split computation across multiple cores and achieve superlinear speedup of RRT*. Jacobs et al. radially partition the space into regions to construct portions of the planning tree in parallel and increase the locality of the computation [8]. C-FOREST [9] samples from a bounded region defined by the length of the best known path and cost metric for which the triangle inequality holds. This effective heuristic allows C-FOREST to only generate samples that have the possibility of improving the solution. KPIECE [10] prioritizes cells in a discretized grid for sampling based upon a notion of a cell's importance to solving a difficult portion of the planning problem. The planner of Burns et al. [11] biases samples towards regions of complexity, as defined by a locally weighted regression and active learning, to improve its ability to navigate narrow passages and other complex regions. Akgun et al. [12] use biased sampling to improve convergence towards optimality.

Varadhan et al. [13] eschew random sampling in favor of a deterministic recursive subdivision of free space into star-shaped partitions, which are then used to generate the roadmap. They use a recursive subdivision of space similar to that of a kd-tree [14], which is used in our method.

Sampling-based planners search nearest-neighbor data structures to find connection points for new samples. Cache-efficient data structures have been an area of active research for many years. Both [15] and [16] discuss the construction of a cache-efficient kd-tree for nearest-neighbor searches. They perform a one-time (i.e., "static") construction of the tree using a van Emde Boas layout [17] which preserves locality in hierarchical traversals (e.g., searches) of the tree. Our method requires the tree to be constructed and queried



(a) CPU cache latency (b) CPU cache size

Fig. 2. Example cache hierarchy a typical modern CPU—the same as used in Section V results. (a) Cache hit latency timings for different levels of the CPU cache hierarchy. (b) The cache levels are depicted graphically.

on-the-fly (i.e., "dynamic"), and methods like [18] can be used to convert static trees to dynamic. Yoon et al. [19] apply cache-efficient construction to bounding volume hierarchies (BVH) and describe how the BVH approach can be extended to kd-trees. They, too, use static construction of van Emde Boas layout and exploit access pattern localities typical of BVH applications (e.g., collision detection and ray tracing) and achieve from good to exceptional (26%–2600%) performance boost based upon the cache-efficient layout. Such methods create cache-efficient layouts for generalized searches whereas CARRT* gains cache-efficiency by constraining searches to a region of a kd-tree.

III. Problem Formulation

Let $\mathcal{C}$ be the bounded d-dimensional configuration space of a robot, and let $\mathcal{C}_{\text{free}} \subseteq \mathcal{C}$ be the subspace of $\mathcal{C}$ that is not in collision with any obstacle in the environment. Let $\mathbf{q} \in \mathcal{C}$ denote a configuration of the robot. The inputs $\mathbf{q}_{\text{init}} \in \mathcal{C}_{\text{free}}$ and $\mathcal{Q}_{\text{goal}} \subseteq \mathcal{C}_{\text{free}}$ are the robot's starting configuration and set of goal configurations, respectively.

The objective of the motion planner in this paper is to compute a collision-free path through the configuration space that reaches the goal region while minimizing a user-specified cost function. We define the path as $\Pi : (\mathbf{q}_{\text{init}}, \mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{\text{end}})$ through $\mathcal{C}_{\text{free}}$ where $\mathbf{q}_{\text{end}} \in \mathcal{Q}_{\text{goal}}$.

The computing platform is a CPU with a cache of limited size that provides low latency access to recently used values from RAM. When the CPU finds a value in the cache, it is a *cache hit*. Conversely, when the CPU does not find the value in the cache, it is a *cache miss*. The difference in latency between a cache hit and a cache miss is called the *cache miss penalty*. A performance objective of the planner is to minimize cache miss penalties by maintaining a working dataset that fits in the cache. Fig. 2 shows the sizes and latencies of the cache levels on a typical modern CPU.

As with other sampling-based motion planners, we require several functions as an input to define the planning problem. The function $\texttt{STEER}(\mathbf{q}_1, \mathbf{q}_2)$ returns a new configuration that would be reached when moving from $\mathbf{q}_1$ toward $\mathbf{q}_2$ up to some specified maximum distance. The function $\texttt{FEASIBLE}(\mathbf{q}_1, \mathbf{q}_2)$ returns `false` if the local path from $\mathbf{q}_1$ to $\mathbf{q}_2$ collides with an obstacle or violates a motion constraint and true otherwise. The function $\texttt{COST}(\mathbf{q}_1, \mathbf{q}_2)$ defines the cost associated with moving from $\mathbf{q}_1$ to $\mathbf{q}_2$ and can represent control effort, Euclidean distance, or any problem-specific cost function that can be used with RRT* [2].

IV. The CARRT* Algorithm

At a high level, CARRT* is an iterative algorithm that builds a motion planning tree with a similar strategy to RRT* [2]. The key difference is that, rather than exploring anywhere in configuration space in every iteration, CARRT* focuses on exploring in distinct smaller regions of the configuration space for short periods of time so as to keep the working dataset small enough to fit in the CPU caches. We call the region being sampled the *active sampling region*.

The planner starts by queuing up a sampling region equal to the problem's configuration space bounds. It then dequeues the active sampling region and samples within the region. Once the region reaches a threshold number of configurations, the planner splits the region in half, queues up the two smaller regions, and repeats the process. The region threshold is tuned to keep the working dataset for a region within the CPU's cache.

CARRT*'s approach to repeatedly splitting configuration space regions in half to create smaller regions naturally synergizes with the kd-tree nearest neighbor search data structure. As such, the planner uses a kd-tree that is explicitly integrated with the region-based sampling. Each active and queued sampling region represents the root of a subtree in the kd-tree—the same subtree that will be explored and expanded during sampling.

CARRT* builds a motion planning tree $G = (V, E)$ with a similar strategy to RRT*. The tree is rooted at the robot's initial configuration. The set of vertices V corresponds to feasible configurations. The directed edge list E defines a tree with the best known feasible paths from the initial configuration to the configurations in V. Each iteration of CARRT* randomly samples a configuration from the active sampling region, and if FEASIBLE, adds the sample to V and an edge to E. Then, within a radius around the new sample, the planner rewires edges in E, replacing longer edges with shorter ones while maintaining the above invariants.

Our planner maintains a second graph $G' = (V, E')$ which shares V from the tree in G and has an *undirected* edge list E' of nearest neighbors of each configuration. This graph is used in the rewiring step discussed in Section IV-D.

A. Sampling Region Queue

CARRT*'s outer loop is shown in Algorithm 1. It starts by initializing the data structures and setting the root of the tree to the robot's initial configuration $\mathbf{q}_{\text{init}}$ (line 1). We initialize the sampling region queue Q in line 3 to have a single region with the bounds of the configuration space $[\mathbf{C}_{\min}, \mathbf{C}_{\max}]$.

The priority queue Q ensures even sampling coverage by defining the highest priority region as the region with the lowest *sample density*:

$$\texttt{Density}(\mathbf{r}) = \frac{(\text{samples considered in region } \mathbf{r})}{(\text{volume of region } \mathbf{r})}.$$

Algorithm 1 CARRT*

```
1:  $V \leftarrow \{\mathbf{q}_{\text{init}}\}, E \leftarrow \varnothing, E' \leftarrow \varnothing$
2:  Q ← empty priority queue
    /* “Q top” is the region with highest priority in Q */
3:  add initial region $[\mathbf{C}_{\min}, \mathbf{C}_{\max}]$ to Q
4:  while not done do
5:      $\mathbf{r} \leftarrow$ remove Q top
6:      PlanRegion($\mathbf{r}$)
7:      if ConfigCount($\mathbf{r}$) < (region config limit) then
8:          add $\mathbf{r}$ back to Q
9:      else
10:         $(\mathbf{r}^{\text{left}}, \mathbf{r}^{\text{right}}) \leftarrow$ split $\mathbf{r}$ region in half along $\mathbf{r}_{\text{axis}}$
11:         $\mathbf{r}^{\text{left}}_{\text{sample\_count}} \leftarrow \frac{1}{2}\mathbf{r}_{\text{sample\_count}}$
12:         $\mathbf{r}^{\text{right}}_{\text{sample\_count}} \leftarrow \frac{1}{2}\mathbf{r}_{\text{sample\_count}}$
13:         add $\mathbf{r}^{\text{left}}$ and $\mathbf{r}^{\text{right}}$ to Q
```

In the outer loop, the planner removes the highest priority region from the queue to make it the active sampling region $\mathbf{r}$ (line 5). Using the function PlanRegion($\mathbf{r}$) (Sec. IV-C), the algorithm samples and extends the active sampling region for a short period of time. CARRT* then determines if $\mathbf{r}$ exceeds the threshold tied to the CPU cache size (line 7). If PlanRegion($\mathbf{r}$) terminated before exceeding the threshold, the planner re-queues the region with its increased sample count and thus lower priority (line 8). Otherwise, $\mathbf{r}$ grew to exceed the region limit, and the planner splits it along an axis shared by the kd-tree (Sec. IV-B) and adds each new region to the queue (lines 10–13). Since CARRT* uniformly samples within a region, we assign half the sample count in $\mathbf{r}$ to each of the new regions. With half the samples, and half the volume, the new regions have the same sample density as $\mathbf{r}$. If, after splitting a region, the resulting child regions still have the highest priority, the planner immediately dequeues one of the new regions and avoids the cache miss penalties that would result from moving to a different region.

Each iteration of the outer loop removes one region from the queue and adds one or two new regions back. Hence, the queue will never be empty at the beginning of each iteration.

B. Integrated KD-Tree

For efficient nearest neighbor searches, CARRT* uses a kd-tree that is integrated with the region-based sampling. A kd-tree is a hierarchical space-partitioning data structure in which branch nodes successively subdivide regions of space by hyperplanes [14], [20]. The subdivisions on the path from the root to any node in the kd-tree define an implicit bounding box for a node. In CARRT*, a kd-tree node's bounding box also represents a sampling region of $\mathcal{C}$-space—it may be the active sampling region, a queued sampling region, a previously split region, or a region that may be queued in the future.

Algorithm 2 adds a configuration $\mathbf{q}$ to the kd-tree. The kd-tree nearest neighbor search (Nearest($\mathbf{q}$)) and fixed-radius nearest neighbor search (Near($\mathbf{q}, \mathbf{r}$)) follow a similar traversal strategy.

Algorithm 2 `KD_Insert`($\mathbf{q}$)

```
1:  $[\mathbf{c}_{\min}, \mathbf{c}_{\max}] \leftarrow [\mathbf{C}_{\min}, \mathbf{C}_{\max}]$
2:  $\mathbf{n} \leftarrow$ kd_root
3:  while $\mathbf{n}_{\text{config}}$ is not nil do
4:      $\mathbf{n}_{\text{size}} \leftarrow \mathbf{n}_{\text{size}} + 1$
5:      axis $\leftarrow$ next axis
6:      split $\leftarrow \frac{1}{2}(\mathbf{c}_{\min}[\text{axis}] + \mathbf{c}_{\max}[\text{axis}])$
7:      if $\mathbf{q}[\text{axis}] <$ split then
8:          if $\mathbf{n}_{\text{left}}$ is nil then
9:              $\mathbf{n}_{\text{left}} \leftarrow$ new node with $_{\text{config}} =$ nil
10:         $\mathbf{n} \leftarrow \mathbf{n}_{\text{left}}$
11:         $\mathbf{c}_{\max}[\text{axis}] \leftarrow$ split
12:     else
13:         follow $\mathbf{n}_{\text{right}}$ similar to $\mathbf{n}_{\text{left}}$ above,
            updating $\mathbf{c}_{\min}$ instead
14: $\mathbf{n}_{\text{config}} \leftarrow \mathbf{q}$
```

The bounds of each node are implicitly defined by the bounds of $\mathcal{C}$ and the node's position in the tree. Line 1 copies bounds of $\mathcal{C}$ into $[\mathbf{c}_{\min}, \mathbf{c}_{\max}]$ defining the bounds of the root node. The loop (line 3) traverses one level deeper in the kd-tree at every iteration, each time dividing the bounding box in $\mathbf{c}$ in half by a hyperplane defined along an implicit axis (lines 5, 6). After determining which side of the split to follow (line 7), the algorithm updates the bounding box (lines 11, 13) to reflect the split.

The axis and the split point are defined to be consistent with the splitting done in Algorithm 1. In our approach the axis is (depth of the node) modulo (dimensions of $\mathcal{C}$-space), and the split is at the midpoint of the node/region's bounding box (line 6).

The traversal loop stops once it has found a node in the tree without a configuration (line 3). The algorithm then adds the configuration $\mathbf{q}$ to the tree (line 14) before returning. The terminal node can be generated in one of two places: (1) the `KD_Insert` algorithm when the left or right child node to traverse is `nil` (lines 8, 13), or (2) in Algorithm 1, when a sampling region is split and a region is empty.

The kd-tree tracks the number of configurations in each subtree (line 4) as configurations are added to it. Algorithm 1 uses the subtree's size (and thus the sampling region's size) to determine when a sampling region needs to be split.

C. Planning Within a Region

CARRT* samples the active region using the inner loop of RRT* modified to run in a cache-aware manner, as shown in Algorithm 3. The notable changes from RRT* are: (1) it has additional loop termination conditions necessary to keep the working dataset small enough to fit in the CPU's cache (line 1); (2) it generates samples from a region of the sampling space (line 2); (3) the nearest neighbor ball radius computation uses a region-based approximation of the sample count (lines 7–8); (4) it tracks the sample count (line 3) to compute the sample-density metric used in the priority queue; and (5) the rewiring strategy accounts for samples being added in regions.

Algorithm 3 `PlanRegion`($\mathbf{r}$)

```
1:  while not done
        and ConfigCount($\mathbf{r}$) $<$ (region config limit)
        and not out of time do
2:      $\mathbf{q}_{\text{rand}} \leftarrow$ random sample from $\mathbf{r}$
3:      $\mathbf{r}_{\text{sample\_count}} \leftarrow \mathbf{r}_{\text{sample\_count}} + 1$
4:      $\mathbf{q}_{\text{nearest}} \leftarrow \texttt{Nearest}(\mathbf{q}_{\text{rand}})$
5:      $\mathbf{q}_{\text{new}} \leftarrow \texttt{STEER}(\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{rand}})$
6:      if $\texttt{FEASIBLE}(\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{new}})$ then
7:          $\mathrm{n}_{\text{approx}} \leftarrow \texttt{ConfigCount}(\mathbf{r}) \times \frac{\texttt{Volume(root)}}{\texttt{Volume(r)}}$
8:          $N \leftarrow \texttt{Near}(\mathbf{q}_{\text{new}}, \min\{\gamma\left(\frac{\log \mathrm{n}_{\text{approx}}}{\mathrm{n}_{\text{approx}}}\right)^{1/d}, \eta\})$
9:          $N_{\texttt{feasible}} \leftarrow \{\mathbf{q} \,|\, \mathbf{q} \in N \wedge \texttt{FEASIBLE}(\mathbf{q}, \mathbf{q}_{\text{new}})\}$
10:         $\mathbf{q}_{\min} \leftarrow \operatorname{argmin}_{\mathbf{q} \in N_{\texttt{feasible}}} \texttt{PathCost}(\mathbf{q}) + \texttt{COST}(\mathbf{q}, \mathbf{q}_{\text{new}})$
11:         $E \leftarrow E \cup (\mathbf{q}_{\text{new}}, \mathbf{q}_{\min})$
12:         for all $\mathbf{q}_{\text{near}} \in N_{\texttt{feasible}} \setminus \mathbf{q}_{\min}$ do
13:             $\mathrm{c}'_{\text{near}} \leftarrow \texttt{PathCost}(\mathbf{q}_{\text{new}}) + \texttt{COST}(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}})$
14:             if $\mathrm{c}'_{\text{near}} < \texttt{PathCost}(\mathbf{q}_{\text{near}})$ then
15:                 $E \leftarrow E \setminus (\mathbf{q}_{\text{near}}, \texttt{Parent}(\mathbf{q}_{\text{near}}))$
16:                 $E \leftarrow E \cup (\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$
17:                 $\texttt{CARRT*Update}(\mathbf{q}_{\text{near}})$
18:         $E' \leftarrow E' \cup \{\{\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}}\} \,|\, \mathbf{q}_{\text{near}} \in N_{\texttt{feasible}}\}$
19:         $V \leftarrow V \cup \mathbf{q}_{\text{new}}$
20:         $\texttt{KD\_Insert}(\mathbf{q}_{\text{new}})$
```

The stopping conditions are specified in line 1. The first criterion ("done") represents typical planning termination checks, e.g., a computation time limit or desired plan cost achieved. The second termination criterion of "`ConfigCount`($\mathbf{r}$) $<$ (region config limit)" checks that the number of configurations in the subgraph contained within the region is smaller than the cache-based limit. The third criterion, "not out of time", sets up a time limit to ensure that CARRT* does not work indefinitely in obstructed or disconnected regions as such regions might otherwise never meet the second stopping condition. In the results section, "out of time" limits the number of samples considered in a region to 1024 (8× the region configuration limit), although other criteria, such as elapsed time, may be used.

In line 2, CARRT* generates a sample in the active sampling region—localizing the computation to the region. The planner finds the random sample's nearest neighbor, and computes $\mathbf{q}_{\text{new}}$ as the result of `STEER`ing towards the random sample (line 5). If the path between $\mathbf{q}_{\text{new}}$ and the nearest neighbor is feasible, CARRT* searches for samples in a ball-radius of $\mathbf{q}_{\text{new}}$. The ball-radius from [2] is computed using the dimensionality of the space d, two tunable parameters γ and η, and the number of configurations in the motion planning graph $|V|$. As CARRT* updates different regions of the space at different times, $|V|$ may be inconsistent with the portion of the graph in the active sampling region. In line 7, the algorithm computes an approximation of $|V|$ in the current region based upon the full motion graph size scaled by the volume ratio of the region to the volume of $\mathcal{C}$. The resulting approximation is fed into the ball radius

Algorithm 4 $\texttt{CARRT*Update}(\mathbf{q})$

```
1: children ← queue with $\mathbf{q}$
2: while not children is empty do
3:     $\mathbf{q}$ ← remove first from children
4:     for all $\mathbf{q}_{\text{near}} \mid \{\mathbf{q}_{\text{near}}, \mathbf{q}\} \in E'$ do
5:         $c' \leftarrow \texttt{PathCost}(\mathbf{q}) + \texttt{COST}(\mathbf{q}, \mathbf{q}_{\text{near}})$
6:         if $c' < \texttt{PathCost}(\mathbf{q}_{\text{near}})$ then
7:             $E \leftarrow E \setminus (\mathbf{q}, \texttt{Parent}(\mathbf{q}))$
8:             $E \leftarrow E \cup (\mathbf{q}, \mathbf{q}_{\text{near}})$
9:             append $\mathbf{q}_{\text{near}}$ to children
```

computation (line 8) in place of the full RRT* graph size.

CARRT*, like RRT*, adds the new configuration to G by linking it to the configuration in the ball radius that produces the shortest path (line 10). The planner then rewires the other configurations in the ball-radius through the new configuration if the rewired path is shorter and feasible (lines 11–17).

D. Rewire Update Strategy

Rewiring in RRT* only considers neighboring configurations in the ball-radius of the new sample $\mathbf{q}_{\text{new}}$. When a neighbor $\mathbf{q}_{\text{near}}$ is rewired through $\mathbf{q}_{\text{new}}$, it can create an opportunity for a neighbor of $\mathbf{q}_{\text{near}}$ to be rewired as well (and of the neighbors' neighbors and so on). RRT* will efficiently propagate such a cascade with future random samples generated from $\mathcal{C}$. If CARRT* followed the same update strategy, the cascade would only be percolated after sampling from a sequence of regions, and thus produce a slower convergence to optimality.

CARRT* takes a different rewiring approach than RRT* to account for this cascade behavior, shown in Algorithm 4. This algorithm is invoked from the main sampling loop of CARRT* (see Algorithm 3) every time it rewires an existing node in the RRT* tree to a better path. It performs a breadth-first traversal of the subtree rooted in the rewired node, rewiring as it goes. The traversal is managed by a FIFO queue, initialized to contain only the root of the rewired subtree (line 1). It then repeatedly dequeues the first node until the queue is empty (line 2, 3). For every configuration $\mathbf{q}$ visited by the traversal, the algorithm visits all of $\mathbf{q}$'s previously computed nearest neighbors as stored in E' (line 4). If the neighboring child's path through $\mathbf{q}$ is shorter than its existing path (line 5, 6), it is rewired (line 7, 8) and added to the queue (line 9) to continue the process of percolating the updates through the subtree.

V. Results

We first evaluate the performance impact on nearest neighbor searches using CARRT*'s region-based sampling in an obstacle-free environment. We then compare CARRT* to RRT* in scenarios involving a point robot and the Rethink Robotics Baxter robot performing a task using 7 degrees of freedom (DOF). Plans are computed on an Intel X5670 2.93 GHz 6-core Westmere processor. Each processing core has a 32 KiB L1 data cache, 256 KiB private L2 cache, and 12 MiB shared L3 cache. The cache-line size is 64 bytes. CARRT* is not multi-threaded, and thus only utilizes 1 core of the processor.

A. KD-Tree Cache Impact

We first evaluate the performance impact of the planner's cache-aware sampling strategy on nearest neighbor searches. We create obstacle-free environments for a point robot in 3, 7, and 14 dimensional space. We compute the average time for a nearest neighbor search with $n = 10^3$ to 10^6 configurations in the kd-tree and plot the results in Fig. 3.

With a log scale x-axis and the theoretic $O(\log n)$ performance of searches, we expect to see a straight-line trend on the graph. In the three plots we observe the non-cache-aware approach has an approximately straight-line trend up to n_{t}=10,000–20,000, and then a steeper straight-line trend after. In our implementation, the kd-tree node occupies 32 bytes, and with a 256 KiB L2 cache, the cache can hold 8192 kd-tree nodes. As the height of the tree also grows logarithmically with n, we expect to see a change in the performance trend at twice the L2 cache capacity. The observed n_{t} matches this expectation.

At lower dimensions (Fig. 3 (a)), the cache-aware approach of CARRT* roughly follows the trend line established before the capacity of L2 cache is exceeded—a nearly ideal result. This enables a $3\times$ performance improvement at $n = 10^6$. The cache-aware approach retains an improvement, though diminishing, for higher dimensions (Fig. 3 (b)-(c)).

B. 7 DOF Ball Obstacle

We consider a scenario in which a point robot must move from one corner of a 7 dimensional cube to the opposite corner while avoiding a spherical obstacle placed at the center of the cube. We run both CARRT* and standard RRT* for comparison. The spherical obstacle implies that an optimal plan can only be found in the limit.

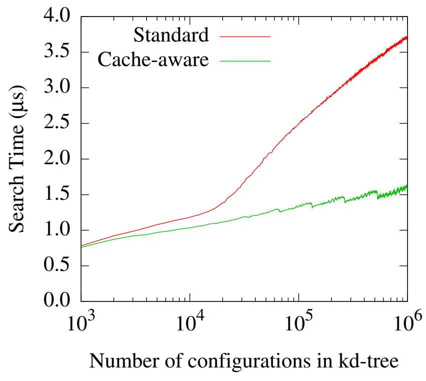
In Fig. 4 (a), we show the average time to run a single nearest neighbor search for a given number of samples in CARRT*'s roadmap. We observe that at approximately 8,000 samples, the performances of RRT* and CARRT* diverge. CARRT*'s nearest neighbor search time always remains below the non-cache-aware RRT* approach.

In Fig. 4 (b), we show the average path cost obtained after running the algorithm a given amount of wall-clock time. On average, CARRT* finds a lower cost plan than RRT* at all times. When viewing Fig. 4 (b) from the perspective of time to reach the same path cost, CARRT* finds a plan at 2.3 s of comparable cost to the plan RRT* finds at 60 s—approximately 26 times faster.

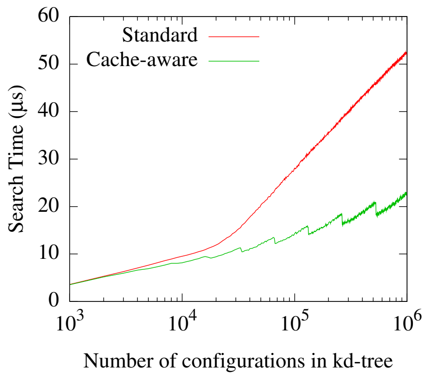
C. Baxter Robot 7 DOF Task

We give a Rethink Robotics Baxter robot the task of moving a book from behind a plant on a shelf to its proper spot on the shelf above, as shown in Fig. 5. The scenario requires the Baxter to move its 7 DOF arm through narrow passages both at the beginning of the task and at the end.

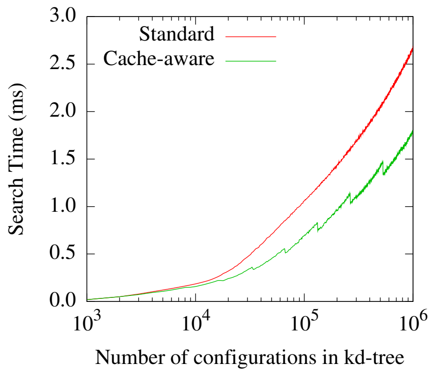
We ran CARRT* and RRT* on the Baxter robot 7 DOF scenario. Fig. 6(a) plots the average nearest neighbor search



(a) 3 DOF kd-tree

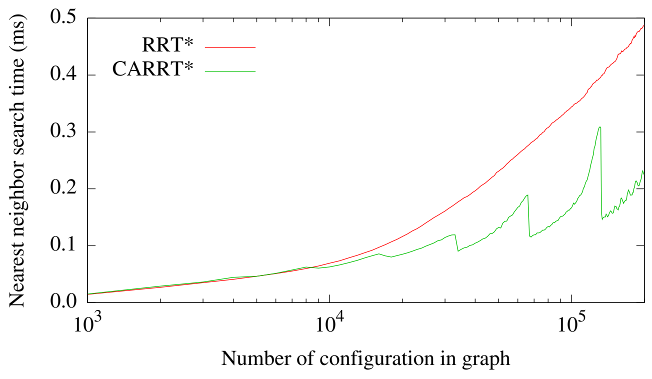


(b) 7 DOF kd-tree

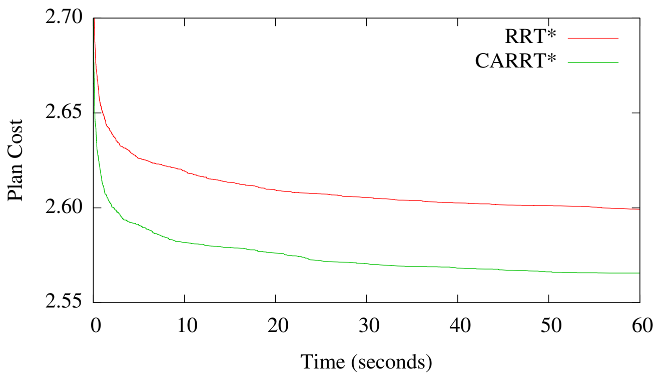


(c) 14 DOF kd-tree

Fig. 3. Average time for a single nearest neighbor search with increasing kd-tree size (n). We search kd-trees using CARRT*'s *cache-aware* region-based sampling and using *standard* uniform random sampling. The kd-tree is 3, 7, and 14 DOF in (a), (b), and (c) respectively, showing the effect of dimensionality on performance. The divergence between $n = 10,000$ and $20,000$ occurs as the tree exceeds the size of the L2 cache.

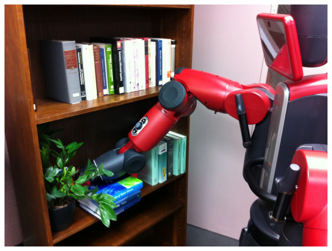


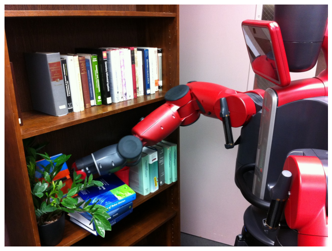
(a) Nearest neighbor search time vs. number of configurations in the kd-tree

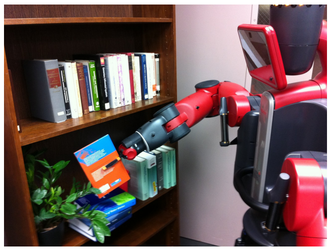


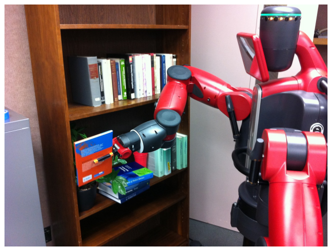
(b) Plan cost vs. CARRT* and RRT* run times

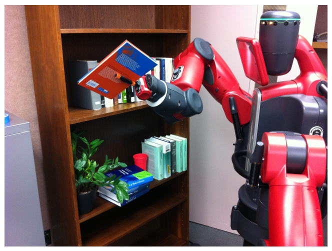
Fig. 4. CARRT* and RRT* compute plans for the 7 DOF ball obstacle scenario. The average time to complete a single nearest neighbor search is shown in (a). The average plan cost computed with a given wall-clock runtime is shown in (b).

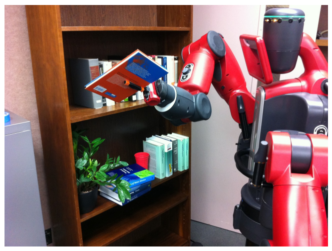

(a)

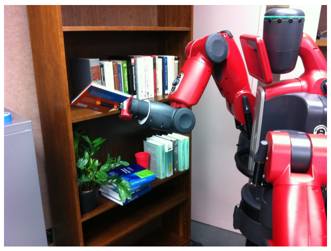

(b)


(c)


(d)


(e)


(f)


(g)

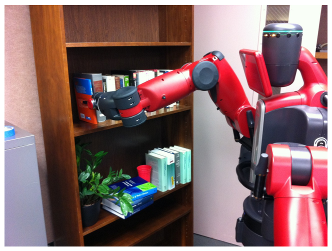
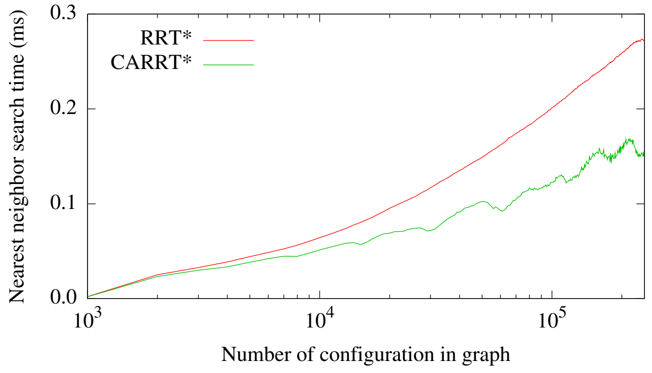
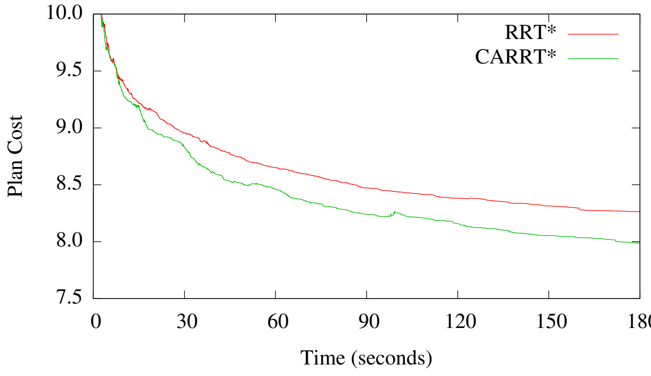

(h)

Fig. 5. The Baxter robot moves a book located behind a plant to its proper place on the shelf above while avoiding obstacles in the cluttered environment. This is a 1-arm, 7 DOF task with a narrow passage to remove the book from behind the plant and another narrow passage to place the book between two books on the shelf above.



(a) Nearest neighbor search time vs. number of configurations in the kd-tree



(b) Plan cost vs CARRT* and RRT* run times

Fig. 6. CARRT* and RRT* compute plans for the Baxter 1-arm 7 DOF scenario. The average time to complete a single nearest neighbor search is shown in (a). The plan cost after a given wall-clock runtime is shown in (b).

time as a function of number of states in the graph, with the x-axis on a log scale. Both CARRT* and RRT* initially start on the same trend line. Between 4,000 and 6,000 samples, RRT* diverges to a slower trend, whereas CARRT* more closely follows the original trend. Fig. 6(b) shows that CARRT* produces lower cost plans faster. CARRT* produces the same plan cost at approximately 90 s as RRT* produces in 180 s, a 2× improvement.

VI. Conclusion

We presented CARRT* (Cache-Aware RRT*), a cache-aware sampling-based asymptotically-optimal motion planner. By progressively partitioning the sampled space into regions that fit into the CPU's cache, CARRT* is able to keep its working dataset for nearest neighbor searches in the CPU cache and avoid delays associated with cache miss penalties. CARRT* also rewires the motion planning graph in a manner that complements the cache-aware subdivision strategy to more quickly refine the motion planning graph toward optimality. We demonstrated the performance benefit of our cache-aware motion planning approach for scenarios with a point robot and the Rethink Robotics Baxter robot.

In future work, we see an opportunity to apply the methods described herein to enable other sampling-based motion planners to be cache-aware. We also plan to investigate methods to auto-tune parameters such as how long a region is explored before being subdivided or queued. These parameters should be based on the properties of the running computing platform. We also plan to address cache-efficiency in other aspects of motion planning, including collision detection.

References

[1] H. Choset, K. M. Lynch, S. A. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005.

[2] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robotics Research*, vol. 30, no. 7, pp. 846–894, June 2011.

[3] D. Levinthal, "Performance analysis guide for Intel® Core™ i7 processor and Intel® Xeon™ 5500 processors," Available: http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2009.

[4] Rethink Robotics, "Baxter Research Robot," http://www.rethinkrobotics.com/products/baxter-research-robot/, 2013.

[5] D. Hsu, G. Sánchez-Ante, and Z. Sun, "Hybrid PRM sampling with a cost-sensitive adaptive strategy," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*. IEEE, 2005, pp. 3874–3880.

[6] S. Rodriguez, S. Thomas, R. Pearce, and N. M. Amato, "RESAMPL: A region-sensitive adaptive motion planner," in *Algorithmic Foundation of Robotics VII*. Springer, 2008, pp. 285–300.

[7] J. Ichnowski and R. Alterovitz, "Parallel sampling-based motion planning with superlinear speedup." in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, 2012, pp. 1206–1212.

[8] S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. Thomas, and N. M. Amato, "A scalable method for parallelizing sampling-based motion planning algorithms," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*. IEEE, 2012, pp. 2529–2536.

[9] M. Otte and N. Correll, "C-FOREST: Parallel shortest path planning with superlinear speedup," *IEEE Trans. Robotics*, vol. 29, no. 3, pp. 798–806, 2013.

[10] I. Şucan and L. E. Kavraki, "A sampling-based tree planner for systems with complex dynamics," *IEEE Trans. Robotics*, vol. 28, no. 1, pp. 116–131, 2012.

[11] B. Burns and O. Brock, "Sampling-based motion planning using predictive models," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*. IEEE, 2005, pp. 3120–3125.

[12] B. Akgun and M. Stilman, "Sampling heuristics for optimal motion planning in high dimensions," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*. IEEE, 2011, pp. 2640–2645.

[13] G. Varadhan and D. Manocha, "Star-shaped roadmaps-a deterministic sampling approach for complete motion planning." in *Robotics: Science and Systems*, vol. 173, 2005.

[14] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sept. 1975.

[15] P. K. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley, "Cache-oblivious data structures for orthogonal range searching," in *Proceedings of the nineteenth annual symposium on Computational geometry*. ACM, 2003, pp. 237–245.

[16] L. Arge, G. Brodal, and R. Fagerberg, "Cache-oblivious data structures," *Handbook of Data Structures and Applications*, vol. 27, 2005.

[17] P. van Emde Boas, "Preserving order in a forest in less than logarithmic time and linear space," *Information processing letters*, vol. 6, no. 3, pp. 80–82, 1977.

[18] J. L. Bentley and J. B. Saxe, "Decomposable searching problems i. static-to-dynamic transformation," *Journal of Algorithms*, vol. 1, no. 4, pp. 301–358, 1980.

[19] S.-E. Yoon and D. Manocha, "Cache-efficient layouts of bounding volume hierarchies," in *Computer Graphics Forum*, vol. 25, no. 3. Wiley Online Library, 2006, pp. 507–516.

[20] S. Maneewongvatana and D. M. Mount, "It's okay to be skinny, if your friends are fat," in *Center for Geometric Computing 4th Annual Workshop on Computational Geometry*, 1999.