RTRBench: A Benchmark Suite for Real-Time Robotics

Mohammad Bakhshalipour
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
bakhshalipour@cmu.edu

Maxim Likhachev
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
maxim@cs.cmu.edu

Phillip B. Gibbons
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
gibbons@cs.cmu.edu

***Abstract*—The emergence of "robotics in the wild" has triggered a wave of recent research in hardware and software to boost robots' compute capabilities. Nevertheless, research in this area is hindered by the lack of a comprehensive benchmark suite.**

In this paper, we present *RTRBench*, a benchmark suite for robotic kernels. *RTRBench* includes 16 kernels, spanning the entire software pipeline of a wide swath of robots, all implemented in C++ for fast execution.

Together with the suite, we conduct an evaluation of the workloads at the *architecture* level. We pinpoint the sources of inefficiencies in a modern robotic processor when executing the robotic kernels, along with the opportunities for improvements.

The source code of the benchmark suite is available in `https://cmu-roboarch.github.io/rtrbench/`.

***Index Terms*—Robotics, Benchmarking, Workload Characterization, Computer Architecture, Simulation.**

I. Introduction

Robots are increasingly playing a prominent role in our technological society. The global robotics market is estimated to reach US $210 billion by 2025, up from $40 billion in 2017 [86]. Accordingly, the global competition to develop the most sophisticated robots in the world is already underway [24], [95]. The path towards developing the most advanced robots in various fields like autonomous vehicles, search and rescue, organ transplant, home assistance, unmanned aerial vehicles, and so forth, has given growing importance to research in this area.

The widespread deployment of "robotics in the wild" necessitates that robots operate effectively and safely under real-time constraints. Hence, robots need to have great compute capabilities to solve various complex artificial intelligence (AI) problems at speed. This requirement has sparked recent research in software and hardware techniques to accelerate various robotic kernels.

Unfortunately, the lack of a comprehensive benchmark suite significantly hampers the research in this emerging area. Most recent research proposals include only one [57], [64], [88] or a few [31], [74] kernels in their evaluations. However, different robotic tasks have different characteristics and requirements: when evaluating a system- or architecture-level technique on only one kernel, its effect on other kernels remains unclear.

In this paper, we present *Real-Time Robotics Benchmark (RTRBench)*, a benchmark suite for robotic workloads. We implement a comprehensive set of kernels that span the whole software pipeline of most autonomous robots. *RTRBench* includes kernels from robot perception, planning, and control. Unlike most prior proposals that use Python, we write all codes in C++ for fast execution. Even though Python modules, which are constituents of prior Python-based suites, have been highly optimized, their performance is still far from their C++-based counterparts [56].

Importantly, to evaluate new hardware techniques, kernels should be easy to simulate on micro-architectural simulators, ahead of any hardware fabrication. We implement a harness for kernels to streamline the simulation process. The harness communicates with the simulator and controls the simulation process.

Finally, we study the architectural implications of the benchmarks running on a modern robotic processor. We pinpoint the sources of inefficiencies in the architecture and discuss the improvement opportunities.

II. Related Work

Robotic workload characterizations of prior work [74], [94] are perhaps the closest work to *RTRBench*. *PerceptIn* [5], a self-driving car startup, details the execution statistics of different kernels internal to their autonomous cars in a recent report [94]. *RoBoX* [74], a hardware acceleration research proposal, evaluates multiple in-house robotic kernels and reports their execution characteristics. Unfortunately, their workloads are not publicly available.

Robotic Operating System (ROS) [7] is a *middleware* for robot development. It provides a framework for operations like low-level device control, hardware abstraction, and package management. It also includes the implementation of some commonly-used robot kernels. Kernels (ROS processes) can be combined to model various robots. ROS provides particular API and communication primitives for enabling such combinations to model a variety of real-world robots. *ROS*, however, does not consider performance as the main objective. The main goal of *ROS* is to provide easy and fast robot development. More than three-fourths of the codes are written in Python, and even those written in C++ are not tuned for performance. Moreover, its primitives like TCP-based inter-process communication present significant challenges for simulating the kernels.

Several pieces of prior work have proposed benchmarks for *particular* robotic tasks. For example, *SBPL* [8] provides a benchmark for search-based robot planning; *OMPL* [9] targets sampling-based motion planning algorithms; *MAVBench* [19] provides a framework for developing micro aerial vehicles;

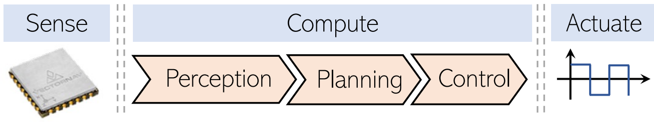


Fig. 1: Robots' computation pipeline.

and *RLBench* [43] is a suite for robot learning kernels. Each of these benchmarks covers a limited range of kernels and does not represent the entire software pipeline of robots. Noteworthy, the combination of these suites, in order to have a more comprehensive set of diverse kernels, is not straightforward, as they use dissimilar set-ups (e.g., Python versus C++). Moreover, many of these suites, if not all, do not accomplish *RTRBench*'s goals: (*i*) real-time performance and (*ii*) easy to simulate.

Finally, some *educational* libraries provide open-source implementations of robotic kernels. For example, the popular *PythonRobotics* library [76] provides a Python code collection of robotic algorithms. These libraries, however, do not consider performance as the main objective, and hence, cannot be used as benchmarks for evaluating techniques in the context of real-time robotics.

In a nutshell, *RTRBench* offers three important features that prior proposals lack wholly or partially:

1) **Comprehensive:** *RTRBench* covers the entire pipeline of a variety of robots, with kernels implementing perception, planning, and control tasks. Many prior proposals (e.g., [8], [9], [43]) include only one stage of the software pipeline.
2) **Real-Time:** *RTRBench* includes kernels implemented for fast execution. From the chosen algorithms down to programming and compilation, *RTRBench* considers performance as the main objective. Prior benchmarks (e.g., [1], [76]) sacrifice performance for implementation ease.
3) **Easy-to-simulate:** *RTRBench* implements kernels such that they are easy to simulate by current micro-architectural simulators (details in §VI). Most prior proposals do not offer this feature; for example, the Python runtime of [7], [43], [76], or the TCP-based communication primitives of [7] pose significant challenges to current simulators.

III. Background: Robot Software Pipeline

Fig. 1 shows the software pipeline of a generic robot. The pipeline consists of *Perception*, *Planning*, and *Control* stages.

A. Perception: The perception unit is responsible for understanding the state of the environment and the robot itself. It reads *raw* data from sensors and *infers* the robot's state (e.g., location, orientation) and the surrounding environment (e.g., obstacles around the robot). Understanding the robot state is known as *localization* and understanding the environment is known as *mapping*.

B. Planning: The planning stage is responsible for generating a path from the current position towards a target position. The planner uses the perception stage's output to comprehend the position of the obstacles and searches the environment to find an efficient (e.g., short), *collision-free* path.

C. Control: The control stage is responsible for generating commands to follow the path generated by the planning stage. The controller calculates the appropriate dynamics (e.g., velocity, acceleration) the robot needs in order to observe to *efficiently* follow the path. Once the dynamics are calculated, the controller sends the proper signals to the robot's actuators.

Depending on the robot, task, and environment, any of the stages could be the performance bottleneck. For example, with a home assistant robot trying to find a soda in a cluttered refrigerator, the perception (understanding the contents of the refrigerator) could be the performance bottleneck. With a pilotless drone trying to find a short path in an environment with high resolution, the planning could limit the end-to-end performance. Finally, with a self-driving car needing a smooth trajectory, the control stage could be the performance bottleneck.

IV. Simulation Methodology

For simulation experiments, we use the *zsim* [77] micro-architectural simulator and model a processor whose specifications resemble the *Intel Core i3-8109U* [11]. Intel Core i3-8109U is a state-of-the-art low-end processor used in robotic systems like the *LoCoBot* manipulator [4] that we will study in this paper.

The processor has two cores, operates at a 3 GHz frequency, and has a 4 MB on-chip cache. Two LPDDR3-2133 memory channels establish processor-memory communications, providing up to 37.5 GB/s bandwidth.

We simulate all kernel programs until they finish and report the results only for the region of interest (ROI). For every kernel, we provide a harness that is used to supply inputs to the kernel, indicate its ROI, and communicate with the simulator.

Finally, we report the evaluation results for every kernel running it with a typical, realistic configuration, on a representative inputset. However, we have implemented all of the kernels in a flexible way such that they can be easily executed with other configuration parameters and inputsets.

V. *RTRBench* Kernels

Table I summarizes *RTRBench*'s kernels along with their key characteristics. We select kernels such that the suite covers the entire software pipeline of most autonomous robots. As an example, in robots operating in low-dimensional spaces (e.g., a self-driving car operating in a 2D/3D space), best-first graph search algorithms like $A^\star$ [40] are used to accomplish path planning. However, in high-dimensional spaces (e.g., a stationary robotic arm with multiple degrees-of-freedom), sampling-based algorithms like *RRT* [55] are used for planning. We

Table I: *RTRBench*'s kernels and their key characteristics.

Kernel	Stage	Bottleneck(s)	Kernel	Stage	Bottleneck(s)
`01.pfl`	Perception	Ray-casting	`09.rrtstar`	Planning	Collision detection, nearest neighbor search
`02.ekfslam`	Perception	Matrix operations	`10.rrtpp`	Planning	Collision detection, nearest neighbor search
`03.srec`	Perception	Point cloud operations, matrix operations	`11.sym-blkw`	Planning	Graph search, string manipulation
`04.pp2d`	Planning	Collision detection	`12.sym-fext`	Planning	Graph search, string manipulation
`05.pp3d`	Planning	Collision detection, graph search	`13.dmp`	Control	Fine-grained serialization
`06.movtar`	Planning	Input-dependent	`14.mpc`	Control	Optimization
`07.prm`	Planning	Graph search, L2-norm calculations	`15.cem`	Control	Sort
`08.rrt`	Planning	Collision detection, nearest neighbor search	`16.bo`	Control	Sort

include both kernels in the suite to represent various real-world robots.

Moreover, we consider algorithms and methods whose effectiveness is established in the community. For example, classic yet extensively-used approaches like particle filter localization [28], whose effectiveness is widely established, are included in our suite. However, recently proposed methods like Q-learning–based path planning has an unclear performance beyond the evaluated scopes, and are not included in our suite.

Following, we provide a description of our kernels, along with their architecture-level evaluations. Noteworthy, while we evaluate the kernels in the context of a simulation framework, they can be employed in scopes beyond simulation, including in ROS-like middlewares and real-world robots. Finally, the kernels' names have two parts: the first part indicates the corresponding pipeline stage and the second part is an abbreviation of the corresponding algorithm/method.

`01.pfl`

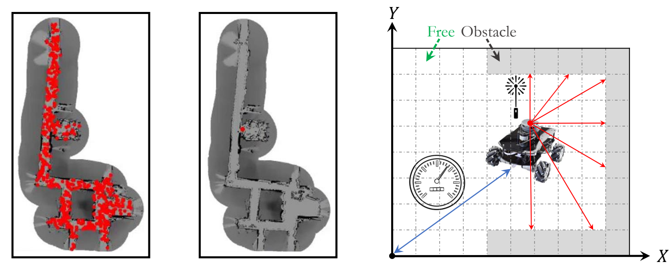
Description: *Particle filter localization* [48], [90], [96] is a method to estimate a robot's state (location, orientation) as it moves and senses the environment, *given a known map*.

Fig. 2 shows an overview of the kernel in an environment modeling a robot moving in the Wean Hall building of Carnegie Mellon University. The robot is equipped with an *odometer* and a *laser rangefinder*.

The method maintains many *particles*, each representing *a particular hypothesis of the robot's state*. All particles are initially sampled from a uniform random distribution, meaning the robot could be anywhere in the environment (Fig. 2-(a)). Throughout the operation, the particles are *re-sampled* based on sensory data: particles whose hypothesis matches the sensed data re-appear with a higher chance. Finally, the particles converge toward the robot's actual state (Fig. 2-(b)).

The odometer measures the distance traveled by the robot at each step (the blue arrow in Fig. 2-(c)). The odometry readings are used to update particles' hypothesis of the robot's state. The laser rangefinder casts rays in different directions and measures the closest obstacle in every direction (the red arrows in Fig. 2-(c)). The laser readings are used to update particles' hypothesis of the obstacles' position. We evaluate the kernel in five different parts of the building.

Evaluation: Our evaluations show that *ray-casting* is the single major performance bottleneck: 67% to 78% of the entire execution time is spent in ray-casting. Ray-casting is the process of *matching* laser readings with hypotheses. Every



(a) Particles in the beginning **(b)** Particles after convergence **(c)** Odometry (blue) and laser (red) measurements in the wild

Fig. 2: Particle filter localization.

particle traverses the map in different directions corresponding to the actually cast rays, and finds the closest obstacle to the robot in every direction. Then, it matches up the traverse distance (hypothesis) with the sensed data from the laser rays, and updates the hypothesis according to a sensor model.

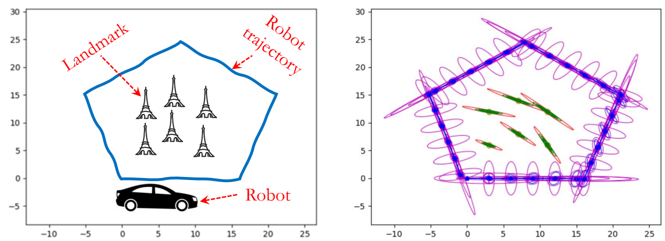
Ray-casting exhibits significant *spatial locality* and *fine-grained parallelism*. The map traversal entails checking the map cells that are nearby each other (spatial locality); also, the cells can be checked in parallel (fine-grained parallelism). These two features make *hardware acceleration* a perfect fit for ray-casting, as realized in Intel's new design: Intel offers a ray-casting accelerator in 10 nm CMOS [46] for edge robotics and augmented reality applications.

`02.ekfslam`

Description: When the environment map is not known, which is a common case for applications like self-driving cars and pilotless drones, the robot should *simultaneously* infer both the surrounding environment and its own location. This operation is known as simultaneous localization and mapping (SLAM). The environment is typically inferred by identifying several *landmarks* (e.g., a tall tower in a city) and keeping track of the robot's state relative to them.

Extended Kalman filter (EKF) is a widely-used method to solve the SLAM problem [52], [91], [97]. *EKF* uses a series of measurements (e.g., the robot's distance from a tower measured using GPS), and infers the state of the robot and the environment. An important feature of *EKF* is its robustness against measurement noises, which is achieved by accounting for *uncertainties* in estimations.

Fig. 3 shows an overview of the kernel in an environment modeling a robot moving through a synthetic setting with six landmarks. The robot constantly reads its distance and angle with the landmarks from its sensors. We add Gaussian-distributed noise to each sensor measurement. Fig. 3-(b) shows the results of *EKF*. Green points are the estimated locations of the landmarks (mapping), and the blue points are the estimated locations of the robot (localization). Red ellipses around the locations represent the uncertainties the method accounts for in its estimations.



(a) The modeled environment **(b)** EKF results for the modeled problem

Fig. 3: SLAM using *Extended Kalman Filter*.

Evaluation: Frequent matrix operations (multiplication, inversion), performed for *updating* the estimations based on sensory data, are the major performance bottleneck of the workload, taking more than 85% of execution time. More specifically, instruction level parallelism (ILP) is limited by the number of function units (FU) that conduct the matrix operations; we observe a decent performance improvement with increasing the number of FUs. However, increasing FUs is not an appealing approach for low-end processors, like the modeled one. As the matrices are not too large[1] and fit in the caches, parallel near-cache computation methods [69] seem a promising approach for performance improvement.

`03.srec`

Description: Scene reconstruction [50], [51], [61], [84] is the process of capturing the shape and appearance of the objects in an environment. We implement the scene reconstruction mechanism of [50], a real-time 3D reconstruction mechanism in *dynamic* scenes. It uses the *iterative closest point (ICP)* algorithm of prior work [66] to reconstruct the scene from different *point clouds*.

A point cloud is a set of data points in space that represents a 3D shape or object. In scene reconstruction [50], the robot's cameras generate multiple different scans of the environment (e.g., with different camera rotations), and then the robot uses *ICP* to evaluate their clouds of points. *ICP* essentially tries to *reconcile* two clouds of points to have a unified understanding of the environment.

We evaluate the kernel using the `living_room` inputset from the ICL-NUIM [39] dataset. Fig. 4-(a) shows the environment (one photo out of all taken by the robot's camera), and Fig. 4-(b) shows the output of *ICP*.

Evaluation: The memory system is a significant bottleneck of the workload. Manipulating point clouds generates numerous *irregular* accesses, overwhelming the memory system. More than 68% of the execution time is spent waiting for memory. *Prefetching* predicted memory accesses in order to reduce memory latency stalls does *not* seem to be a promising solution because (*i*) the memory accesses are not easy-to-predict, and (*ii*) the bottleneck is memory bandwidth, not memory latency. Near-data processing approaches [65] seem more fitting, particularly because of the low compute-to-communicate ratio [60] of data.

[1]The size of matrices is proportionate to the *number of different measurement types* (*distance* and *angle* in the modeled application).

(a) The environment **(b)** The reconstructed scene

Fig. 4: 3D reconstruction in dynamic scenes.

Another important bottleneck is massive matrix operations (e.g., cross-multiplication, inversion). Though matrix data has a regular layout that is amenable to high ILP, the operations would need a large number of FUs to exploit the ILP.

Finally, a GPU, if it could be afforded in the robot, is a by far better platform for scene reconstruction. GPUs offer significantly higher memory bandwidth, tolerate memory stalls to a large extent, and can better exploit the data-parallel nature of scene reconstruction [27].

`04.pp2d`

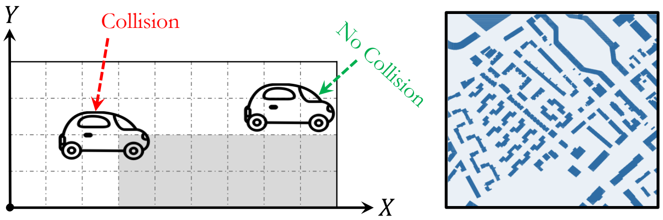
Description: Path planning is the process of finding an *efficient*, *collision-free* path from the current state (location) to a goal state for a robot in complex surroundings.

In path planning, the environment is represented as a graph (Fig. 5-(a)), and the planner *searches* it using a graph search algorithm. $A^\star$ [40], along with its variants and extensions, is the seminal algorithm widely used in various robot path planning applications. The key novelty of $A^\star$ over other graph search algorithms like Dijkstra is its *heuristic* for estimating the distance from the goal. We use *Euclidean distance* as the heuristic function. The search algorithm returns the path that should be taken by the robot to reach the goal.

To ensure the final path is collision-free, the planner performs frequent *collision detection* operations (Fig. 5-(a)). Collision detection is the task of checking whether the robot would collide with obstacles in the environment if it were in a particular state.

We implement a mobile robot navigating in 2D environments, modeling a self-driving car navigating in a city. We use `Boston_1_1024` of Moving AI [87], which is a snapshot of Boston, Massachusetts, as the environment (Fig. 5-(b)). The car's length×width is $4.8^m \times 1.8^m$. We choose the start and goal points such that the car traverses a long distance, observing different obstacle patterns.

Evaluation: Collision detection is the major performance bottleneck. More than 65% of the entire execution time is spent in collision detection. Similar to ray-casting (§V.1),



(a) Planning and collision detection **(b)** The environment

Fig. 5: 2D path planning.

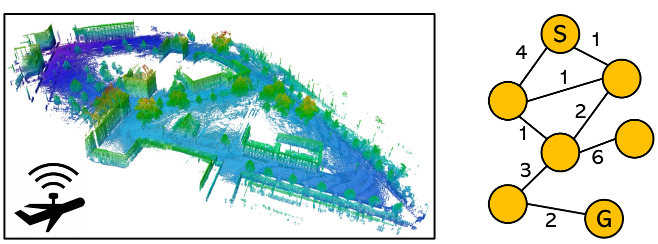
collision detection exhibits significant *fine-grained parallelism* and *spatial locality*.

Checking the collision status of every part of the robot's body is independent of other parts; the operations can be completely parallelized. Importantly, the parallelism is extremely fine-grained: every operation *is simply checking a cell value*. Also, the parts of the robot that are tested for collision belong to *one integrated body*; collision detection computation is fundamentally spatially-located: the occupancy grid cells that are checked during a collision detection are nearby each other.

Significant fine-grained parallelism and spatial locality make hardware acceleration a perfect fit for collision detection, as realized by recent work [16], [57], [62], [63].

`05.pp3d`

Description: We implement a mobile robot navigating in a 3D environment. The kernel is similar to **`pp2d`**, but the planning has one more dimension: the z dimension. We model an unmanned aerial vehicle (UAV), a.k.a. drone, navigating in an outdoor environment, `fr_campus` of [2], which is a snapshot of Freiburg campus (Fig. 6-(a)). We assume the UAV is small and fits in one resolution unit. Like **`pp2d`**, we choose the start and goal points such that the UAV traverses a long distance, observing different obstacle patterns.



(a) The modeled environment **(b)** Path graph search

Fig. 6: 3D path planning.

Evaluation: Other than collision detection, the graph search is another major performance bottleneck. Fig. 6-(b) shows an example of the graph search problem. Search algorithms like Dijkstra and $A^\star$ try to find the shortest path between a start point (e.g., 'S' in Fig. 6-(b)) and a goal point (e.g., 'G' in Fig. 6-(b)). These algorithms (*i*) exhibit *irregular* traversal, and (*ii*) are hard to parallelize. As a result, the execution suffers from tremendous serialization in both intra-node (limited ILP due to load misses) and inter-node (limited thread-level parallelism due to data dependency) computations.

Irregular-data prefetchers can reduce the data stalls to some extent. We evaluated an over-approximated implementation of VLDP [83] and found that it can eliminate around one-third of the data misses. Also, *speculative parallelism* approaches [13], [16], [45] could be quite effective in parallelizing such hard to parallelize graph search algorithms. Another appealing approach is *data-centric execution*. Particularly because the computation on every graph node is short (e.g., heuristic calculation, cost update), data-centric architectures, that offload short tasks to different execution engines located near the corresponding data [58], could significantly accelerate the search process.

`06.movtar`

Description: This kernel represents a complex planning problem, in which a robot is trying to catch a *moving* target (Fig. 7). The assumption is that the robot knows the trajectory of the target (i.e., the location of the target at any given *time*). The environment is 2D but path planning is done in 3D, with *time* as the third dimension.

We create our own synthetic environments. Every location in the environment has a particular *cost* for the robot. The goal of the robot is to catch the target with minimum cost.

Without a well-informing heuristic, this problem cannot be solved in a reasonable amount of time in large environments. We use *backward Dijkstra* [17] as our heuristic function: before starting planning, the backward Dijkstra algorithm is executed to calculate the heuristic values in an environment-aware manner (e.g., accounting for obstacles).

After calculating the heuristic values, the search algorithm runs on a conceptual 3D graph to catch the moving target with the lowest possible cost. We use *Weighted* $A^\star$ (*WA**) [72] instead of $A^\star$ to accelerate the graph search. *WA** *inflates* the heuristic by a factor of ε. This way, the search is biased towards the nodes that are closer to the goal, resulting in a faster search. On the flip side, the final path cost could become ε times higher than the shortest path cost.

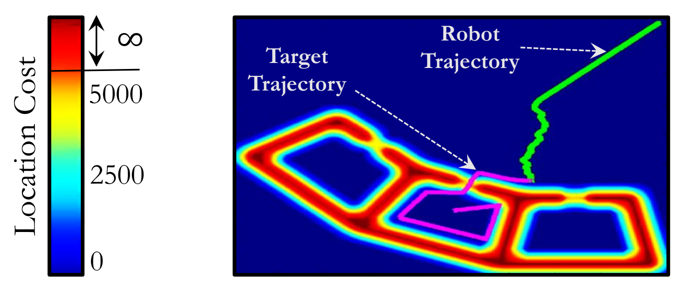


Fig. 7: Catching a moving target.

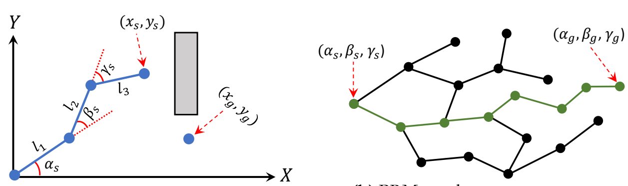
Evaluation: The performance of the kernel is largely dependent on the inputset. In large environments, the kernel exhibits virtually the same characteristics as **`pp3d`**. In small environments, however, unlike **`pp3d`**, the contribution of the heuristic calculation latency to the end-to-end latency grows up to 62%. *Approximate hardware acceleration* [30], [59] can be used for improving the performance of heuristic calculations.

Heuristic values, particularly when tight *optimality guarantees* are not required, are amenable for approximation.

07.prm

Description: Motion planning for stationary robotic arm manipulators with multiple degrees-of-freedom (DoF) is one of the challenging, time-consuming kernels in robotics. The problem has been targeted in a variety of levels from algorithm to, recently, architecture [57], [62], [63], [64].

Fig. 8-(a) shows an example of arm planning. A 3-DoF robot should move its *end-effector* (end of the robot's arm) from a start point, (x_s, y_s), to a goal point, (x_g, y_g). Planning for a robotic arm is performed in joints' angle space: the planner calculates a series of $(\alpha_i, \beta_i, \gamma_i)$s to guide the end-effector from the start point to the goal point.



(a) 3-DoF robotic arm **(b)** PRM graph

Fig. 8: Robotic arm motion planning.

Robotic arm planning has as many *dimensions* as its DoF. When the number of dimensions grows, it is not feasible to include the entire configuration space in the graph. For example, for a 5-DoF robotic arm with a minimum angle rotation of $10°$, the configuration space could include $(\frac{360°}{10°})^5 \approx 60M$ different values. Building a graph with that many nodes would make the problem infeasible to solve in a reasonable time.

High-dimensional planning is performed by *sampling* the configuration space. *Probabilistic RoadMap (PRM)* [14], [25], [49], [78] is a seminal algorithm for path planning in high-dimensional configuration spaces. *PRM* has offline and online phases. In the offline phase, it takes *random samples* from the configuration space of the robot, then tests whether they are collision-free, and finally connects *nearby* samples to form a graph, an example of which is shown in Fig. 8-(b).

In the online phase, *PRM* adds the start and goal configurations to the graph, and accomplishes the planning by searching the graph with an algorithm like $A^\star$ to find a path from the start to the goal (green path in Fig. 8-(b)). We model a 5-DoF arm manipulator operating in two synthetic environments, as shown in Fig. 9. `Map-F` represents a free environment, and `Map-C` shows a cluttered one.

Evaluation: The offline process could be significantly lengthy, but it is paid only once and is done offline. The online search process, however, is on the critical path and can limit the performance. The search suffers from the same problems as in **pp2d**, even more so. The samples are literally random, and the graph traversal is quite irregular. Moreover, the data of every node is even larger, as every node keeps the entire sample configuration (n floating point numbers corresponding to n joint angles; e.g., 40 bytes with a 5-DoF arm). Therefore, the importance of prefetching is more pronounced in this context.

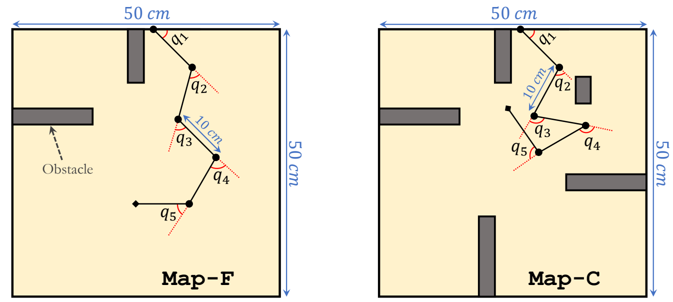


Fig. 9: Synthetic maps for evaluating the robotic arm.

Also, frequent *L2-norm* calculations, which are done to calculate the distance of samples in n-dimension space, is another bottleneck. Prior work [41], [67] proposes *specialized imprecise hardware* for operations like L2-norm and multiply-accumulate, that can be used to accelerate *PRM*.

08.rrt

Description: *PRM* is efficient in *static* environments (i.e., the obstacles around the robot do not change). However, since it relies on an offline-trained graph, it cannot react to changes in the environment: if the obstacles in the environment are relocated, the built graph is out of date.

Rapidly-exploring Random Trees (RRT) [22], [29], [55] is a widely-used algorithm for high-dimensional planning in *dynamic* environments. *RRT* draws random samples and extends a *tree* (not a more general graph) from the start configuration towards the goal configuration. An example of such a tree is shown in Fig. 10. During extending the tree, *RRT* checks the collision status of different configurations with the obstacles in the environment. We evaluate the kernel on `Map-C` and `Map-F`. Unlike **prm**, **rrt** does not have any apriori knowledge of the maps, and hence builds the entire data structure online.

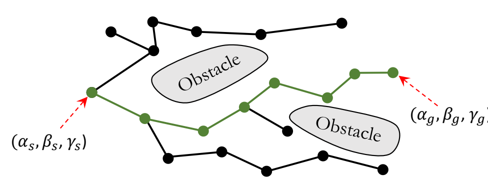


Fig. 10: Arm manipulator planning by the *RRT* algorithm.

Evaluation: Collision detection is a major performance bottleneck of the application, taking up to 62% of the execution time. Unlike *PRM* that has an offline phase and performs collision checks offline, *RRT* does not have an offline phase, and hence, collision checks fall into the critical path of the execution. As discussed for **pp2d**, hardware acceleration is able to largely accelerate collision detection, as realized by recent work [16], [57], [62], [63].

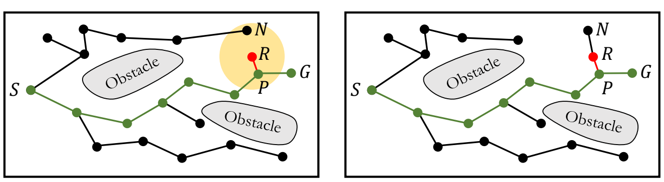
Nearest neighbor search is another performance bottleneck, taking up to 31% of the execution time. When drawing a sample, *RRT* searches the other samples to find the near ones to connect the new sample to them. This operation exhibits irregular memory accesses, because the samples whose values (angles) are close could be allocated in distant memory locations. This results in a large L1 data cache miss ratio (12%-22% in our experiments), significantly hurting the performance. The problem is observed in other classic applications like pattern recognition [89] and computer vision [15], as well. Prior work proposes *in-memory computation* [75] and *informed caching* [71] for accelerating the nearest neighbor search operations.

09.rrtstar

Description: *RRT* is fast but can return an inefficient, costly path [47]. *RRT** [34], [47], [92] is a variant of *RRT* that returns an *asymptotically optimal* path. *RRT** improves path quality by *rewiring* the tree: when a random sample is added to the tree, near neighbors are evaluated and the connections change if the addition of the new node can reduce the path cost.

Fig. 11 shows an example of rewiring. Fig. 11-(a) shows the tree before rewiring. First, a random sample, named R (the red node), is drawn. Then, the nearest neighbor of R in the tree, named P, is found. R is connected to P and becomes its child. The *RRT* algorithm stops at this step. However, *RRT** evaluates the near neighbors of R (the yellow circle) for rewiring. There is only one node, named N, in the neighborhood. *RRT** evaluates whether *removing* the previous connection of N and connecting it to R improves the cost of path to N or not. If so, N is rewired, as shown in Fig. 11-(b).

We evaluate *RRT** on the `Map-C` and `Map-F` environments.



(a) Before rewiring **(b)** After rewiring

Fig. 11: A rewiring example with *RRT**.

Evaluation: *RRT** is significantly slower (up to 8× in our experiments) but generates shorter paths (1.6× on average) as compared to *RRT*.

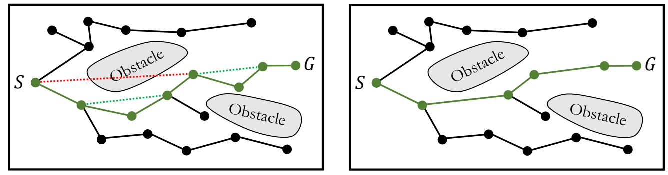
RRT*, like *RRT*, suffers from high collision detection and nearest neighbor search latency. The contribution of the latter increases to up to 49% due to frequent rewiring operations.

10.rrtpp

Description: *RRT** provides an asymptotically optimal path but it can significantly increase the execution time of *RRT*.

Prior work [32], [68], [81], [93] proposes *post-processing* the path produced by *RRT* to improve the path cost, while avoiding the high execution time of *RRT**. The post-processing involves iterating over the nodes of the path and *shortcutting* them to reduce the final cost. Fig. 12 shows examples of shortcutting.

Fig. 12-(a) shows the path before post-processing. The post-processing works based on the *triangle inequality*. Two nodes along the path are shortcutted if they can be directly connected to each other; i.e., there are not any obstacles among them. For example, in Fig. 12-(a), the two node pairs connected by dashed green lines can be shortcutted, while the node pair connected by a dashed red line cannot. Fig. 12-(b) shows the path after post-processing. The post-processing step could run for several iterations to further reduce the path cost.



(a) Before post-processing **(b)** After post-processing

Fig. 12: Post-processing the path found by *RRT*.

We evaluate *RRT** on the `Map-C` and `Map-F` environments.

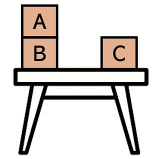
Evaluation: *RRT* with post-processing exhibits computation characteristics (and path cost) that lie in between *RRT** and the baseline *RRT*. The overhead of nearest neighbor search operations decreases as compared to *RRT** due to the lack of rewiring operations; and, the cost of post-processing is added on top of the baseline *RRT*.

11.sym-blkw

Description: Symbolic planning [18], [21], [35] is a general *framework* to solve a variety of robotic planning problems. In symbolic planning, the problem is represented using high-level, human-readable symbols. The inputs of the planner are the valid symbols, initial state, goal state, and valid *actions*. An action is a set of operations done by the robot and results in changing the state of the robot and/or environment. Every action has *preconditions* and *effects*. Preconditions are the conditions a state must have for an action to be applicable to it. Effects are the changes an action makes to a state. The problem is ultimately represented as a graph search and the planner computes *a sequence of actions* to reach the goal state from the initial state. The strength of symbolic planning is its generality: one *symbolic planner* can solve any problem that can be described in the symbolic language.

We implement a symbolic planner and solve the *blocks world* problem [38] in its context. Fig. 13-(a) shows parts of a symbolic representation of the blocks world problem, and Fig. 13-(b) shows a sketch of the problem in its initial state. Even though blocks world is a toy problem, it shares the same kernel with many realistic NP-hard search problems including robotic vision, motor control, and probabilistic inference [85].

```
Symbols: A, B, C, Table
Initial conditions: On(A, B), On(B, Table), On(C, Table), ...
Goal conditions: On(B, C), On(C, A), On(A, Table)
Actions:
        Move(b, x, y)
        Preconditions: On(b, x), Clear(b), Clear(y), ...
        Effects: On(b, y), Clear(x), !On(b, x), !Clear(y)
        ⋮
```



(a) Symbolic description of a blocks world problem **(b)** Blocks world

Fig. 13: Blocks world problem.

Evaluation: The kernel has only dominant operations: searching the graph nodes to find a set of actions, and *string manipulation* inside nodes. The former exhibits the same behavior as other graph search kernels in the context of robot planning; e.g., **pp2d**, **pp3d**, and **prm**.

The string manipulation has long been targeted in the context of computer architecture [12], [33] for classic applications like packet routing and web querying. With the growing popularity of applications like bioinformatics and genome sequence analysis, and the viability of hardware acceleration, *string manipulation hardware accelerators* are revisited by recent work [20], [37]. Such accelerators can be repurposed for accelerating symbolic planning, with minimum effort.

12.sym-fext

Description: We model a firefighting problem and solve it in the context of symbolic planning. The problem is inspired by the final challenge at MIT's 1st Summer School on Cognitive Robotics [10]. There are two robots: a mobile robot and a quadcopter. By landing on the mobile robot, the quadcopter pours water on the fire. The quadcopter has a limited battery level and a limited water tank; in case of low battery or water, the quadcopter needs to charge its battery or fill its tank before pouring water on the fire. The ultimate goal of the problem is to extinguish the fire. Fig. 14 shows parts of the symbolic representation of the problem.

```
Symbols: A, B, C, D, E, W, F, Q, R
Initial conditions: Quad(Q), Rob(R), At(Q, B), At(R, A), Loc(A), InAir(Q), ...
Goal conditions: ExtThree(F)
Actions:
        MoveToLoc(x,y)
        Preconditions: Loc(x), Loc(y), At(R, x), InAir(Q)
        Effects: At(R, y), !At(R, x)

        FillWater(x)
        Preconditions: Quad(x), OnRob(x), EmptyTank(x), At(R, W), At(Q, W)
        Effects: !EmptyTank(x), FullTank(Q)
        ⋮
```

Fig. 14: Firefighter robots.

Evaluation: The kernel uses the same symbolic planner as in **sym-blkw**, and hence, it largely exhibits the same (architectural) characteristics. However, **sys-fext** exhibits a higher level of parallelism ($\sim$3.2$\times$) since it has *more valid actions*. Every action translates into an edge in the graph representation of the problem, and the neighbors of every node at every step can be evaluated in parallel.

13.dmp

Description: *Dynamic movement primitives (DMP)* [53], [79], [80] is a control kernel to generate a *smooth* trajectory based on the path computed by the robot's path planner. *DMP* represents the problem using a virtual *spring and damper* system and adapts it to the planned path.

DMP uses Gaussian bias functions and shape parameters to *define* the overall trajectory shape. These parameters are often acquired through *imitation learning* [42] and linear regression, typically through a single demonstration. Once the parameters are acquired, the final trajectory, including the position, velocity, and acceleration parameters, is computed.

We train the model using data gathered from a demonstration of an in-house wheeled robot. We evaluate the model for a reference trajectory depicted by orange in Fig. 15. The black lines in Fig. 15 show the trajectory (left) and velocity (right) generated by *DMP*.

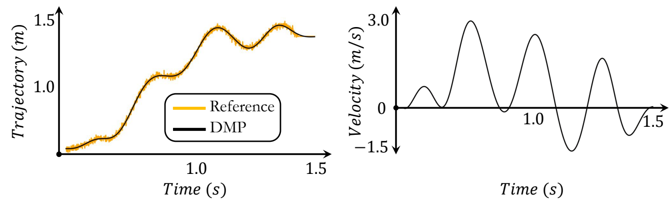


Fig. 15: Dynamic movement primitives.

Evaluation: The ILP is low (instructions-per-cycle (IPC) < 1) due to significant data dependency in the algorithm: the trajectory, velocity, and acceleration are all computed incrementally. *Dataflow* architectures [36] have been shown to be effective for this kind of computation.

14.mpc

Description: *Model predictive control (MPC)* is a mechanism used to control a process while satisfying a set of constraints. In robotics, it is used to generate control inputs to the robot's actuators to *efficiently* follow the path absorbed from the planning stage [23], [26], [54]. For example, with a self-driving car, the constraints could be the maximum speed, and the goal could be following a path with minimum fuel usage.

Fig. 16 shows an overview of the kernel in an environment modeling a self-driving car following a long reference trajectory while *not* exceeding predefined velocity and acceleration values. The cost is formulated as a function of the *deviation* from the reference trajectory and the state *change* during the path.

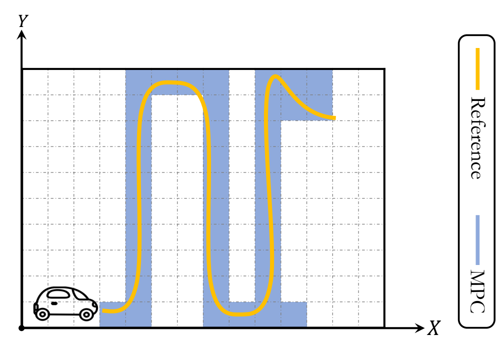


Fig. 16: Model predictive control.

Evaluation: The major bottleneck of the kernel is solving the optimization problem, taking more than 80% of the entire execution time. RoBoX [74] proposes a full-fledged accelerator for accelerating this process. It uses specialized logic and near-data computation to solve the problem significantly faster.

15.cem

Description: We model a ball-throwing robot whose throwing *skills* get improved using *reinforcement learning*. We use

V-REP [73] to simulate the robot and the environment. Fig. 17 shows the environment. A 2-DoF robotic arm applies a certain *force* to throw a ball towards a certain goal. The purpose of reinforcement learning is to learn the best force and configuration (joints' angles). The *reward* of the learning is how close the final location of the ball is to the goal.

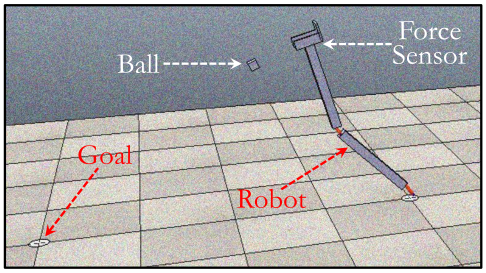


Fig. 17: Ball-throwing robot.

Cross-entropy method (CEM) [70] is a Monte Carlo optimization method. *CEM* learns the *policy* (throwing parameters) by repeatedly drawing samples, collecting rewards, and minimizing the *cross-entropy loss* to shift the policy towards samples that result in larger rewards. We execute *CEM* for five iterations and draw fifteen samples in every iteration. Fig. 18 shows how reward (higher is better) changes over learning.

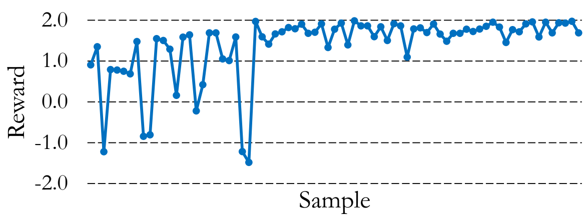


Fig. 18: Rewards over time using *CEM*.

Evaluation: Recent work [82] proposes hardware acceleration for reinforcement learning that can be well applicable in this context as well. Also, we find the *sort* operations (for finding the largest rewards) as a non-trivial execution bottleneck of the algorithm, taking around one-third of the entire execution time, depending on the learning algorithm configuration.

`16.bo`

Description: In robotics, *Bayesian optimization (BO)* is used to optimize control parameters in reinforcement learning [44]. *BO* is data-efficient and gradient-free, and is increasingly used to solve a variety of control problems.

We implement *BO* in the context of the ball-throwing robot scenario. We use an upper confidence bound (UCB) acquisition function. Training and testing are done using a Gaussian process. Fig. 19 shows how the reward changes over the course of the 45 iterations of the learning process.

Evaluation: The kernel exhibits largely the same characteristics as **`cem`**. However, computationally, it is more intensive ($\sim$15000$\times$ more iterations). Hardware acceleration of the reinforcement learning kernel can be a perfect architectural solution to accelerate the application. Also, since more metadata

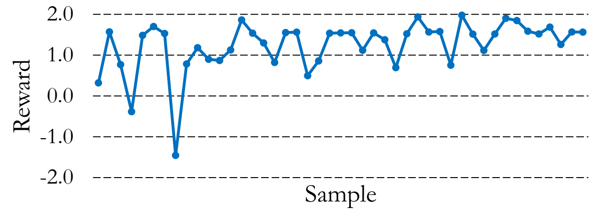


Fig. 19: Rewards over time using *BO*.

is kept with *BO*, its sort operation is more time-consuming ($\sim$6$\times$ as compared to **`cem`**).

VI. Implementation Details

System Requirements: We test *RTRBench* under **`Ubuntu 18.04`** with **`Linux Kernel 4.15`**, and compile with **`GCC 11.1.0`**. However, *RTRBench* can be used with a variety of other operating systems and compilers. As we use **`C++17`**, the minimum required **`GCC`** version is **`8.0`** (**`Clang`**: **`5.0`**; **`Visual Studio`**: **`15.8`**).

Also, we integrate the kernels with *zsim* [77], and communicate the regions of interest (ROIs) using *zsim hooks*. Without *zsim* (either real execution or in the context of other simulators), the harness instructions will be *safely* executed: no effect on correctness and virtually zero effect on performance. We believe *RTRBench* can be smoothly integrated/used with other simulators, as well; however, the ROI communications should be coded based on the target simulator.

Usage & Flexibility: We provide a Makefile for every kernel. Also, we provide a usage help message for every kernel. Running the binary file of a kernel with `--help` will print the help message. For example, Fig. 20 shows an example of a help message.

```
$ ./rrt.out --help

USAGE:
  ./rrt.out  [OPTIONS] [FLAGS]

OPTIONS:
  --bias <val>         Random number generation bias
  --config <val>       Input config file
  --epsilon <val>      Epsilon (minimum movement)
  --map <val>          Input map file
  --output <val>       Output path file
  --radius <val>       Neighborhood distance
  --samples <val>      Maximum samples

FLAGS:
  --help, -h           Print help message
```

Fig. 20: An example of a help message.

Also, as the figure shows, we implement the kernels in a completely flexible manner; all of the configuration/execution parameters can be set/changed from the command line. Not shown in the figure, we provide proper default values for the configuration parameters.

Inputsets: In the paper, we typically report kernel execution results for one inputset per kernel. However, in the repository, we provide multiple inputsets for many of the kernels.

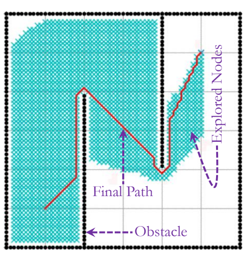
VII. Comparison with Open-Source Repositories

As we mentioned in §II, there are a few educational open-source libraries that provide implementations of robotic kernels. The main problem with these libraries is that they do not consider performance as the main objective, and hence, cannot be used as a benchmark for real-time robotics.

In this section, we compare the performance of our suite against *PythonRobotics (P-Rob)* [6], [76] and *CppRobotics (C-Rob)* [1]. *P-Rob* is a popular robotic library with $\sim$4.8*K* forks and $\sim$14.8*K* stars (as of 04/01/2022). *P-Rob* provides a Python code collection of the robotic kernels operating on small, synthetic inputsets. *C-Rob* translates some of the *P-Rob* kernels to C++.

As a case study, we compare **`pp2d`** with the counterpart kernels in *P-Rob* (**`a_star.py`**) and *C-Rob* (**`a_star.cpp`**). We removed the code for generating animations from the competitor libraries, to accelerate their execution. We conduct this experiment on a real machine, as the competitor libraries are not easy to simulate (Python runtime, etc.). Our machine uses Intel Xeon CPU E5-2670 [3] cores operating at 2.60 GHz, with the operating system and compiler described in §VI.

As inputset, we use the map provided by the competitor libraries, which is depicted in Fig. 21-(a). Because the map is small, we also scale it by different factors to evaluate the implementations in larger (or finer-resolution) environments. Fig. 21-(b) shows the execution time results.



(a) The environment

Scale	Time (s)		
	P-Rob	C-Rob	RTRBench
1	1.44E-01	3.00E-02	4.03E-04
2	6.49E-01	2.70E-01	1.37E-03
4	3.53E+00	2.15E+00	5.04E-03
8	2.07E+01	2.92E+01	2.13E-02
16	1.38E+02	4.37E+02	1.03E-01
32	9.93E+02	6.56E+03	4.83E-01
64	7.65E+03	6.56E+03	2.20E+00

(b) Execution time of different methods

Fig. 21: Performance comparison of different libraries.

As the results show, the competitor libraries are far from real-time. Our implementation is 357×–3469× faster than *P-Rob*, and 74×–13576× faster than *C-Rob*. *P-Rob*, not surprisingly, suffers from the tremendous overhead of the Python runtime. For *C-Rob*, we investigated its source code for this particular kernel, and found that the main source of inefficiency is passing large data structures to functions needlessly *by value* instead of *by reference*. As noted above, and highlighted by this performance comparison, these libraries are designed for educational purposes and not for real-time experiments.

VIII. Conclusion

Research on real-time robotics is significantly hindered by the lack of a comprehensive benchmark suite. In this paper, we present *RTRBench*, a benchmark suite for robotic kernels. *RTRBench* includes 16 kernels, spanning the entire software pipeline of a wide swath of robots. Together with the suite, we conduct an evaluation of the workloads at the architecture level and suggest opportunities for performance improvements.

Acknowledgment

This work was supported in part by National Science Foundation grant CCF-2028949, by a VMware University Research Fund Award, and by the Parallel Data Lab (PDL) Consortium (Alibaba, Amazon, Datrium, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Salesforce, Samsung, Seagate, and TwoSigma). Mohammad Bakhshalipour was supported by the Apple CMU ECE PhD Fellowship in Integrated Systems. We would like to thank the anonymous reviewers for their valuable comments, and our shepherd, Jaekyu Lee, for his feedback.

References

[1] "CppRobotics," https://github.com/onlytailei/CppRobotics.

[2] "Freiburg Campus 360 Degree 3D Scans," http://ais.informatik.uni-freiburg.de/projects/datasets/fr360/.

[3] "Intel Xeon Processor E5-2670," https://ark.intel.com/content/www/us/en/ark/products/64595.

[4] "LoCoBot: An Open Source Low Cost Robot," http://www.locobot.org/.

[5] "PerceptIn," https://www.perceptin.io/.

[6] "PythonRobotics," https://github.com/AtsushiSakai/PythonRobotics.

[7] "ROS - Robot Operating System," https://www.ros.org/.

[8] "Search-Based Planning Lab," http://www.sbpl.net/.

[9] "The Open Motion Planning Library," http://ompl.kavrakilab.org/.

[10] "1st Summer School on Cognitive Robotics," http://cognitive-robotics17.csail.mit.edu/, 2017.

[11] "Intel Core I3-8109U Processor," https://ark.intel.com/content/www/us/en/ark/products/135936, 2018.

[12] M. Aldwairi, T. Conte, and P. Franzon, "Configurable String Matching Hardware for Speeding Up Intrusion Detection," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 1, pp. 99–107, 2005.

[13] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "Perspective: A Sensible Approach to Speculative Automatic Parallelization," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020, pp. 351–367.

[14] D. Baek, M. Hwang, H. Kim, and D.-S. Kwon, "Path Planning for Automation of Surgery Robot Based on Probabilistic Roadmap and Reinforcement Learning," in *International Conference on Ubiquitous Robots (UR)*. IEEE, 2018, pp. 342–347.

[15] F. Bajramovic, F. Mattern, N. Butko, and J. Denzler, "A Comparison of Nearest Neighbor Search Algorithms for Generic Object Recognition," in *International Conference on Advanced Concepts for Intelligent Vision Systems*. Springer, 2006, pp. 1186–1197.

[16] M. Bakhshalipour, S. B. Ehsani, M. Qadri, D. Guri, M. Likhachev, and P. B. Gibbons, "RACOD: Algorithm/Hardware Co-Design for Mobile Robot Path Planning," in *International Symposium in Computer Architecture (ISCA)*. IEEE/ACM, 2022.

[17] G. Bakhtyar, V. Nguyen, M. Cetin, and D. Nguyen, "Backward Dijkstra Algorithms for Finding the Departure Time Based on the Specified Arrival Time for Real-Life Time-Dependent Networks," *Journal of Applied Mathematics and Physics*, vol. 4, no. 1, 2016.

[18] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas, "Symbolic Planning and Control of Robot Motion [grand Challenges of Robotics]," *IEEE Robotics & Automation Magazine*, vol. 14, no. 1, pp. 61–70, 2007.

[19] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. Reddi, "MAVBench: Micro Aerial Vehicle Benchmarking," in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2018, pp. 894–907.

[20] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand *et al.*, "Genasm: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2020, pp. 951–966.

[21] G. Canal, E. Pignat, G. Alenyà, S. Calinon, and C. Torras, "Joining High-Level Symbolic Planning with Low-Level Motion Primitives in Adaptive HRI: Application to Dressing Assistance," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 3273–3278.

[22] X. Cao, X. Zou, C. Jia, M. Chen, and Z. Zeng, "RRT-Based Path Planning for an Intelligent Litchi-Picking Manipulator," *Computers and Electronics in Agriculture*, vol. 156, pp. 105–118, 2019.

[23] A. Carron, E. Arcari, M. Wermelinger, L. Hewing, M. Hutter, and M. N. Zeilinger, "Data-Driven Model Predictive Control for Trajectory Tracking with a Robotic Arm," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3758–3765, 2019.

[24] M. Chan, "China and the U.S Are Fighting a Major Battle Over Killer Robots and the Future of AI," *TIME*, Sep 2019. [Online]. Available: https://time.com/5673240/china-killer-robots-weapons

[25] G. Chen, N. Luo, D. Liu, Z. Zhao, and C. Liang, "Path Planning for Manipulators Based on an Improved Probabilistic Roadmap Method," *Robotics and Computer-Integrated Manufacturing*, vol. 72, p. 102196, 2021.

[26] E. Dantec, R. Budhiraja, A. Roig, T. Lembono, G. Saurel, O. Stasse, P. Fernbach, S. Tonneau, S. Vijayakumar, S. Calinon *et al.*, "Whole Body Model Predictive Control with a Memory of Motion: Experiments on a Torque-Controlled Talos," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 8202–8208.

[27] S. Darabi, N. Mahani, H. Baxishi, E. Yousefzadeh-Asl-Miandoab, M. Sadrosadati, and H. Sarbazi-Azad, "NURA: A Framework for Supporting Non-Uniform Resource Accesses in GPUs," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–27, 2022.

[28] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte Carlo Localization for Mobile Robots," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 1999, pp. 1322–1328.

[29] J. Denny, R. Sandström, A. Bregger, and N. M. Amato, "Dynamic Region-Biased Rapidly-Exploring Random Trees," in *Algorithmic Foundations of Robotics XII*. Springer, 2020, pp. 640–655.

[30] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2012, pp. 449–460.

[31] Y. Feng, B. Tian, T. Xu, P. Whatmough, and Y. Zhu, "Mesorasi: Architecture Support for Point Cloud Analytics Via Delayed-Aggregation," in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2020, pp. 1037–1050.

[32] D. Ferguson, N. Kalra, and A. Stentz, "Replanning with RRTs," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2006, pp. 1243–1248.

[33] E. Fernandez, W. Najjar, and S. Lonardi, "String Matching in Hardware Using the FM-Index," in *International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 218–225.

[34] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed RRT*: Optimal Sampling-Based Path Planning Focused Via Direct Sampling of an Admissible Ellipsoidal Heuristic," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2014, pp. 2997–3004.

[35] C. R. Garrett, T. Lozano-Perez, and L. P. Kaelbling, "FFRob: Leveraging Symbolic Planning for Efficient Task and Motion Planning," *The International Journal of Robotics Research*, vol. 37, no. 1, pp. 104–136, 2018.

[36] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019, pp. 199–213.

[37] V. Y. Gudur and A. Acharyya, "Hardware-Software Codesign Based Accelerated and Reconfigurable Methodology for String Matching in Computational Bioinformatics Applications," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 17, no. 4, pp. 1198–1210, 2018.

[38] N. Gupta and D. S. Nau, "On the Complexity of Blocks-World Planning," *Artificial Intelligence*, vol. 56, no. 2-3, pp. 223–254, 1992.

[39] A. Handa, T. Whelan, J. McDonald, and A. J. Davison, "A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 1524–1531.

[40] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[41] J. Huang, J. Lach, and G. Robins, "A Methodology for Energy-Quality Tradeoff Using Imprecise Hardware," in *Design Automation Conference (DAC)*. IEEE, 2012, pp. 504–509.

[42] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation Learning: A Survey of Learning Methods," *ACM Computing Surveys*, vol. 50, no. 2, pp. 1–35, 2017.

[43] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison, "RLBench: The Robot Learning Benchmark & Learning Environment," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3019–3026, 2020.

[44] N. Jaquier, L. Rozo, S. Calinon, and M. Bürger, "Bayesian Optimization Meets Riemannian Manifolds in Robot Learning," in *Conference on Robot Learning*. PMLR, 2020, pp. 233–246.

[45] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A Scalable Architecture for Ordered Parallelism," in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2015, pp. 228–241.

[46] M. Kar, A. Agarwal, S. Hsu, D. Moloney, G. Chen, R. Kumar, H. Sumbul, P. Knag, M. Anders, H. Kaul *et al.*, "A Ray-Casting Accelerator in 10nm CMOS for Efficient 3D Scene Reconstruction in Edge Robotics and Augmented Reality Applications," in *Symposium on VLSI Circuits*. IEEE, 2020, pp. 1–2.

[47] S. Karaman and E. Frazzoli, "Incremental Sampling-Based Algorithms for Optimal Motion Planning," *Robotics Science and Systems VI*, vol. 104, no. 2, 2010.

[48] P. Karkus, D. Hsu, and W. S. Lee, "Particle Filter Networks with Application to Visual Localization," in *Conference on Robot Learning*. PMLR, 2018, pp. 169–178.

[49] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[50] M. Keller, D. Lefloch, M. Lambers, S. Izadi, T. Weyrich, and A. Kolb, "Real-Time 3D Reconstruction in Dynamic Scenes Using Point-Based Fusion," in *International Conference on 3D Vision (3DV)*. IEEE, 2013, pp. 1–8.

[51] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction," *ACM Transactions on Graphics (ToG)*, vol. 36, no. 4, pp. 1–13, 2017.

[52] V. Kokovkina, V. Antipov, V. Kirnos, and A. Priorov, "The Algorithm of EKF-SLAM Using Laser Scanning System and Fisheye Camera," in *Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO)*. IEEE, 2019, pp. 1–6.

[53] L. Koutras and Z. Doulgeri, "Dynamic Movement Primitives for Moving Goals with Temporal Scaling Adaptation," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 144–150.

[54] F. Künhe, J. Gomes da Silva, and W. Fetter Lages, "Mobile Robot Trajectory Tracking Using Model Predictive Control," in *II IEEE Latin-American Robotics Symposium (LARS)*, vol. 51. Citeseer, 2005.

[55] S. M. LaValle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," Iowa State, Ames, IA, USA, Tech. Rep. TR 98-11, 1998.

[56] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's Plenty of Room at the Top: What Will Drive Computer Performance After Moore's Law?" *Science*, vol. 368, no. 1079, June 2020.

[57] S. Lian, Y. Han, X. Chen, Y. Wang, and H. Xiao, "Dadu-P: A Scalable Accelerator for Robot Motion Planning in a Dynamic Environment," in *Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[58] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-Centric Computing Throughout the Memory Hierarchy," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020, pp. 417–433.

[59] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration," in *International Symposium in Computer Architecture (ISCA)*. ACM/IEEE, 2016, pp. 66–77.

[60] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting Locality in Graph Analytics Through Hardware-Accelerated Traversal Scheduling," in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2018, pp. 1–14.

[61] Z. Murez, T. van As, J. Bartolozzi, A. Sinha, V. Badrinarayanan, and A. Rabinovich, "Atlas: End-To-End 3D Scene Reconstruction from Posed Images," in *Computer Vision–ECCV 2020: 16th European Conference*. Springer, 2020, pp. 414–431.

[62] S. Murray, W. Floyd-Jones, G. Konidaris, and D. J. Sorin, "A Programmable Architecture for Robot Motion Planning Acceleration," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 185–188.

[63] S. Murray, W. Floyd-Jones, Y. Qi, D. J. Sorin, and G. D. Konidaris, "Robot Motion Planning on a Chip," in *Robotics: Science and Systems*, 2016.

[64] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The Microarchitecture of a Real-Time Robot Motion Planning Accelerator," in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2016, pp. 1–12.

[65] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A Modern Primer on Processing in Memory," *arXiv preprint arXiv:2012.03112*, 2020.

[66] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "Kinectfusion: Real-Time Dense Surface Mapping and Tracking," in *International Symposium on Mixed and Augmented Reality*. IEEE, 2011, pp. 127–136.

[67] L. Ni, H. Huang, and H. Yu, "On-Line Machine Learning Accelerator on Digital RRAM-Crossbar," in *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2016, pp. 113–116.

[68] T. Nishi and T. Sugihara, "Motion Planning of a Humanoid Robot in a Complex Environment Using RRT and Spatiotemporal Post-Processing Techniques," *International Journal of Humanoid Robotics*, vol. 11, no. 02, p. 1441003, 2014.

[69] A. V. Nori, R. Bera, S. Balachandran, J. Rakshit, O. J. Omer, A. Abuhatzera, B. Kuttanna, and S. Subramoney, "REDUCT: Keep It Close, Keep It Cool!: Efficient Scaling of DNN Inference on Multi-Core CPUs with Near-Cache Compute," in *International Symposium in Computer Architecture (ISCA)*. ACM/IEEE, 2021, pp. 167–180.

[70] L. Petrović, J. Peršić, M. Seder, and I. Marković, “Cross-Entropy Based Stochastic Optimization of Robot Trajectories Using Heteroscedastic Continuous-Time Gaussian Processes,” *Robotics and Autonomous Systems*, vol. 133, p. 103618, 2020.

[71] R. Pinkham, S. Zeng, and Z. Zhang, “QuickNN: Memory and Performance Optimization of k-d Tree Based Nearest Neighbor Search for 3D Point Clouds,” in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 180–192.

[72] I. Pohl, “Heuristic Search Viewed As Path Finding in a Graph,” *Artificial intelligence*, vol. 1, no. 3-4, pp. 193–204, 1970.

[73] E. Rohmer, S. P. Singh, and M. Freese, “V-REP: A Versatile and Scalable Robot Simulation Framework,” in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2013, pp. 1321–1326.

[74] J. Sacks, D. Mahajan, R. C. Lawson, and H. Esmaeilzadeh, “RoBoX: An End-To-End Solution to Accelerate Autonomous Control in Robotics,” in *International Symposium in Computer Architecture (ISCA)*. ACM/IEEE, 2018, pp. 479–490.

[75] J. Saikia, S. Yin, Z. Jiang, M. Seok, and J.-s. Seo, “K-Nearest Neighbor Hardware Accelerator Using In-Memory Computing SRAM,” in *International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2019, pp. 1–6.

[76] A. Sakai, D. Ingram, J. Dinius, K. Chawla, A. Raffin, and A. Paques, “PythonRobotics: A Python Code Collection of Robotics Algorithms,” *arXiv preprint arXiv:1808.10703*, 2018.

[77] D. Sanchez and C. Kozyrakis, “ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems,” in *International Symposium in Computer Architecture (ISCA)*. ACM/IEEE, June 2013.

[78] R. M. C. Santiago, A. L. De Ocampo, A. T. Ubando, A. A. Bandala, and E. P. Dadios, “Path Planning for Mobile Robots Using Genetic Algorithm and Probabilistic Roadmap,” in *International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*. IEEE, 2017, pp. 1–5.

[79] S. Schaal, “Dynamic Movement Primitives-A Framework for Motor Control in Humans and Humanoid Robotics,” in *Adaptive Motion of Animals and Machines*. Springer, 2006, pp. 261–280.

[80] S. Schaal, J. Peters, J. Nakanishi, and A. Ijspeert, “Control, Planning, Learning, and Imitation with Dynamic Movement Primitives,” in *Workshop on Bilateral Paradigms on Humans and Humanoids: IEEE International Conference on Intelligent Robots and Systems (IROS 2003)*, 2003, pp. 1–21.

[81] J. H. Seo, H. Lee, and K.-D. Kim, “A Parallelization Algorithm for Real-Time Path Shortening of High-DOFs Manipulator,” *IEEE Access*, vol. 9, pp. 123 727–123 741, 2021.

[82] S. Shao, J. Tsai, M. Mysior, W. Luk, T. Chau, A. Warren, and B. Jeppesen, “Towards Hardware Accelerated Reinforcement Learning for Application-Specific Robotic Control,” in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–8.

[83] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, “Efficiently Prefetching Complex Address Patterns,” in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2015, pp. 141–152.

[84] D. Shin, Z. Ren, E. B. Sudderth, and C. C. Fowlkes, “3D Scene Reconstruction with Multi-Layer Depth and Epipolar Transformers,” in *International Conference on Computer Vision*. IEEE, 2019, pp. 2172–2182.

[85] J. Slaney and S. Thiébaux, “Blocks World Revisited,” *Artificial Intelligence*, vol. 125, no. 1-2, pp. 119–153, 2001.

[86] Statista Research Department, “Global Robotics Market Revenue 2018–2025,” https://www.statista.com/statistics/760190/worldwide-robotics-market-revenue/, 2021.

[87] N. R. Sturtevant, “Benchmarks for Grid-Based Pathfinding,” *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, vol. 4, no. 2, pp. 144–148, 2012.

[88] T. Tan, R. Weller, and G. Zachmann, “SIMDop: SIMD Optimized Bounding Volume Hierarchies for Collision Detection,” in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 7256–7263.

[89] B. Tang and H. He, “ENN: Extended Nearest Neighbor Method for Pattern Recognition,” *IEEE Computational Intelligence Magazine*, vol. 10, no. 3, pp. 52–60, 2015.

[90] I. Ullah, Y. Shen, X. Su, C. Esposito, and C. Choi, “A Localization Based on Unscented Kalman Filter and Particle Filter Localization Algorithms,” *IEEE Access*, vol. 8, pp. 2233–2246, 2019.

[91] I. Ullah, X. Su, X. Zhang, and D. Choi, “Simultaneous Localization and Mapping Based on Kalman Filter and Extended Kalman Filter,” *Wireless Communications and Mobile Computing*, vol. 2020, pp. 2 138 643:1–2 138 643:12, 2020.

[92] W. Xinyu, L. Xiaojuan, G. Yong, S. Jiadong, and W. Rui, “Bidirectional Potential Guided RRT* for Motion Planning,” *IEEE Access*, vol. 7, pp. 95 046–95 057, 2019.

[93] E. Yoshida, K. Yokoi, and P. Gergondet, “Online Replanning for Reactive Robot Motion: Practical Aspects,” in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2010, pp. 5927–5933.

[94] B. Yu, W. Hu, L. Xu, J. Tang, S. Liu, and Y. Zhu, “Building the Computing System for Autonomous Micromobility Vehicles: Design Constraints and Architectural Optimizations,” in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2020, pp. 1067–1081.

[95] A. Zaleski, “China’s Blueprint to Crush the US Robotics Industry,” *CNBC*, Sep 2017. [Online]. Available: https://www.cnbc.com/2017/09/06/chinas-blueprint-to-crush-the-us-robotics-industry.html

[96] Q.-b. Zhang, P. Wang, and Z.-h. Chen, “An Improved Particle Filter for Mobile Robot Localization Based on Particle Swarm Optimization,” *Expert Systems with Applications*, vol. 135, pp. 181–193, 2019.

[97] B. Zheng and Z. Zhang, “An Improved EKF-SLAM for Mars Surface Exploration,” *International Journal of Aerospace Engineering*, vol. 2019, 2019.