RACOD: Algorithm/Hardware Co-design for Mobile Robot Path Planning

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

Seyed Borna Ehsani
University of Washington
Seattle, Washington, USA
behsani@uw.edu

Mohamad Qadri
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
mqadri@andrew.cmu.edu

Dominic Guri
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
dguri@andrew.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

RACOD is an algorithm/hardware co-design for mobile robot path planning. It consists of two main components: *CODAcc*, a hardware accelerator for *collision detection*; and *RASExp*, an algorithm extension for runahead path exploration. *CODAcc* uses a novel MapReduce-style hardware computational model and massively parallelizes *individual* collision checks. *RASExp* predicts future path explorations and proactively computes its collision status ahead of time, thereby overlapping *multiple* collision detections. By affording multiple cheap *CODAcc* accelerators and overlapping collision detections using *RASExp*, *RACOD* significantly accelerates planning for mobile robots operating in arbitrary environments. Evaluations of popular benchmarks show up to 41.4× (self-driving cars) and 34.3× (pilotless drones) speedup with less than 0.3% area overhead.

While the performance is maximized when *CODAcc* and *RASExp* are used together, they can also be used individually. To illustrate, we evaluate *CODAcc* alone in the context of a stationary robotic arm and show that it improves performance by 3.4×–3.8×. Also, we evaluate *RASExp* alone on commodity many-core CPU and GPU platforms by implementing it purely in software and show that with 32/128 CPU/GPU threads, it accelerates the end-to-end planning time by 8.6×/2.9×.

CCS CONCEPTS

• **Hardware → Application-specific VLSI designs**; • **Computer systems organization → Parallel architectures**.

KEYWORDS

hardware acceleration, speculative parallelism, robotics, path planning, collision detection

ISCA '22, June 18–22, 2022, New York, NY, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8610-4/22/06.
https://doi.org/10.1145/3470496.3527383

ACM Reference Format:
Mohammad Bakhshalipour, Seyed Borna Ehsani, Mohamad Qadri, Dominic Guri, Maxim Likhachev, and Phillip B. Gibbons. 2022. RACOD: Algorithm/Hardware Co-design for Mobile Robot Path Planning. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3470496.3527383

1 INTRODUCTION

Path planning is a core task in nearly any autonomous robot. Path planning is the process of finding a *collision-free* path in an environment from the current state (location) to a goal state. *Collision detection* is the task of checking whether the robot would collide with obstacles in the environment if it were in a particular state.

"Path planning can be so compute- and memory-intensive that it is typically off-loaded to the cloud," according to a recent quote by Intel engineers [33]. However, such offloading often cannot meet real-time requirements because the latency of communicating with the cloud is too high and unpredictable [33]. Thus, real-time applications, e.g., an aerial vehicle performing complex maneuvers [7], require path planning to be performed by the robot itself, as fast as possible. Not only does the high cost of planning present performance challenges, it can also make the robot *unsafe*: safety is greatly dependent on how *quickly* the robot can react to emergencies [49]. In fact, the inability to generate plans in real-time is the major barrier that hinders the widespread deployment of robots in the wild [36].

In this paper, we study path planning at the architectural level. We first architect *COllision Detection Accelerator (CODAcc)*, a hardware accelerator for collision detection. Collision detection is extremely time-consuming, taking up to 99% of the entire planning time [11, 30, 31, 34]. *CODAcc* accelerates collision detection by massively parallelizing individual collision checks: different parts of the robot's body are tested for collision in parallel with each other. *CODAcc* achieves a high level of parallelism by employing a novel MapReduce-style collision computation: the memory addresses, corresponding to different locations of the environment that the robot's body would intersect, are generated in parallel using multiple function units; the addresses that are mapped to the same *cache blocks* are tested for collision together.

While *individual* collision checks are perfectly parallelized by *CODAcc*, the end-to-end speedup of hardware acceleration is limited due to the obstructive serialization in the process of exploring the environment (i.e., path *search*). Search algorithms like Dijkstra, $A^\star$, and their variants and extensions exhibit little to no parallelism for path planning [37]. The reason is the inherent serialization in the process of searching for an optimal (or efficient) path. At every step, search algorithms explore a certain location in the environment and determine the movement with the highest prospect of reaching the destination (e.g., move left), *then* assuming that the movement is taken, they explore the new location. In other words, parallelizing the path search algorithms is not straightforward because until the current location is explored, the next to-be-explored location *is not known*. This serialization barrier becomes the single major performance bottleneck of path planning after *CODAcc* has been used to speed up individual collision checks.

We overcome this serialization barrier using a technique named *Run-Ahead State Exploration (RASExp)*. The key idea is to *predict* future states (locations) that will likely be explored, perform their collision checks *speculatively* ahead of time, and *memoize* the collision status for later usage. The key observation is that although the search patterns are too complicated for state-of-the-art hardware predictors [40], they can be *semantically* predicted using domain knowledge. Specifically, as we show in §2.2.2, path planning exhibits "cone-like" patterns: the footprint of explored areas mostly comprises *narrow cones with few turns in direction*. We leverage this observation to predict future states and overlap their collision checks with current collision checks. We equip the system with multiple (up to 32) *CODAcc* accelerators; at every step, we perform collision checks of current (*demand*) and future (*speculative*) states in parallel, thereby achieving additional speedups. We call the combined algorithm/accelerator system *Run-Ahead COllision Detection (RACOD)*.

In summary, our work makes the following contributions:

(A) We architect *CODAcc*, an efficient collision detection hardware accelerator. *CODAcc* is applicable to a wide range of robots operating in *arbitrary* environments. As we will discuss in §7.1, prior work on hardware acceleration of collision detection [31, 35] makes restrictive assumptions about the environment (e.g., (most) obstacles never move), which greatly limits their application.

(B) We propose *RASExp*, a novel algorithm extension for parallelizing path search algorithms. *RASExp* is the first *semantic* speculation technique in path search and beyond.

(C) We evaluate *CODAcc* and *RASExp* both separately and synergistically:

- *CODAcc* accelerates mobile planning by 1.24×–1.49×.
- *RASExp*, with no hardware change, accelerates mobile robot planning by 8.6× (2.9×) on a commodity CPU (GPU).
- *RACOD*, i.e., the synergistic implementation of *CODAcc* and *RASExp*, accelerates mobile robot planning by up to 34.3× (pilotless drone) and 41.4× (self-driving car), with less than 0.3% hardware overhead.
- To show our design's applicability beyond mobile robots, we also evaluate *CODAcc* in the context of a *stationary* robotic arm with a state-of-the-art sampling-based planner. We show that it improves end-to-end planning time by 3.4×–3.8×.

2 BACKGROUND AND MOTIVATION

2.1 Path Planning & Bounding Volumes

Our focus is mobile robots, like self-driving cars and pilotless drones, whose ability to operate in real time is challenged by the lengthy planning time [7]. An example is given in Figure 1 (left). A circle-shaped robot, with radius r, moves from a start point, (x_s, y_s), to a goal point, (x_g, y_g). The path planner's task is to find an optimal (or efficient) collision-free path from the start point to the goal point: a series of (x_i, y_i)s such that if the robot is located at any (x_i, y_i), it will not collide with the obstacles (gray cells) in the environment.

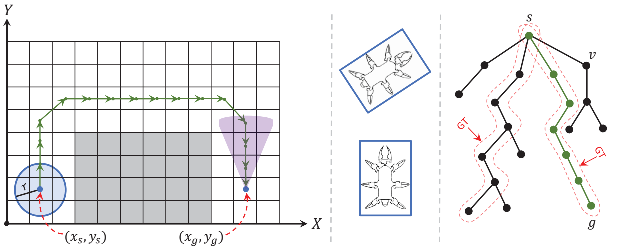

Figure 1: 2D mobile robot path planning (left), oriented bounded box (middle), and path graph search (right).

An *occupancy grid*, produced by the robot's *perception unit*, is provided to the path planner. The occupancy grid indicates which cells in the environment are free ('0'), and which cells are occupied with obstacles ('1'). The perception unit constantly updates the occupancy grid to reflect the most recent understanding of the environment. Note that perception is a separate stage in the robot's software pipeline: the occupancy grid does not change *during* planning.

To ensure the final path is collision-free, the planner performs *collision detection* for the points that are considered for inclusion in the final path. To find out whether a point satisfies this condition, the planner first should calculate *which cells will be involved if the robot is placed at that point*. This operation is known as *forward kinematics (FK)*. Then, the planner checks *all* the cells determined by FK, and if none of them is an obstacle, the point is identified as collision-free.

In the example, if r is around one resolution unit, as the figure suggests, then for every point, collision detection entails checking 9 cells: the point's cell and the 8 cells surrounding it. Collision detection can be quite intensive: e.g., with a radius $r = 10^{cm}$ and a resolution unit of 1^{cm}, collision detection for *any* point entails checking 384 cells.

In practice, the robot's shape can be more complex than a circle. For example, a rough shape of the Arduino Ant Hexapod Robot [3] is shown inside the rectangles in Figure 1 (middle). In such cases, precise computation of FK can itself be too complex and costly, let alone the post-FK collision detection.

Oriented Bounded Box (OBB) [17] is a method used to handle robots with arbitrary shapes and orientations. *OBB* bounds the shape with an oriented rectangle (in 2D; cube in 3D), as exemplified by the blue rectangles in Figure 1 (middle). By bounding the robot's body, collision detection reduces to checking whether the robot's

OBB falls into a collision-free cell or not.

Observation I: Collision detection can be massively parallelized; however, the parallelism is extremely fine-grained.

Checking the collision status of every part of the robot’s body is independent of other parts; the operations can be completely parallelized. E.g., in the example of Figure 1 (left), the 9 cells the robot’s body may occupy at a time can all be checked in parallel. Importantly, the parallelism is extremely fine-grained: every operation *is simply checking a cell value*.

The fine-grained parallelism makes hardware acceleration a perfect fit for collision detection; simply *ORing* the cells in hardware (inherently parallel) will provide the collision status. Vectorization and multithreading may seem like promising alternatives for such computation, but each has its own problems. Vectorization can accelerate the collision detection of *axis-aligned OBB*s, but other orientations would not have a regular, array-like layout in memory, rendering vector instructions useless. Multithreading, in CPU or GPU, is a poor match for this kind of computation: creating, preparing, and joining a thread is much costlier than simply checking a cell value.

Observation II: Collision detection computation exhibits a high level of spatial locality.

Because a robot is *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, clustered around the physical robot. In the Figure 1 (left) example, a 256-byte cache *dedicated* for collision detection memory accesses results in a 99+% hit ratio.

2.2 Path Search

2.2.1 Search Algorithm. Mobile robot path planning is ultimately reduced to a *graph search* problem: nodes are states (locations) and edges are *robot motions*. For example, with a robot that can move in four cardinal directions (N, E, S, and W) in a 2D environment, every non-terminal node is connected to 4 surrounding nodes (4-connected grid). With a robot that can further move in four intercardinal directions (NE, SE, SW, and NW), the graph will be an 8-connected grid.

The graph can be searched using practically any graph search algorithm to extract an optimal or efficient path. $A^\star$ [20], along with its variants and extensions (§5.9), 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 employing a *heuristic* that results in significant speedup, e.g., an *estimate* of a point’s distance from the goal. In what follows, we briefly overview the algorithm (pseudo-code in §3.2.1).

Consider the graph depicted in Figure 1 (right). The algorithm should find a path from the start point (s) to the goal point (g). For every node v in the graph, $A^\star$ defines $f(v) = g(v) + h(v)$, where $g(v)$ is the *actual* movement cost (distance) from s to v, and $h(v)$ is the *heuristic* cost from v to g (an underestimate of the actual cost). In this paper, the default heuristic is *Euclidean distance*.

$A^\star$ maintains an OPEN list, which initially contains only s. At every iteration, the node with the lowest f value is *expanded*: it is removed from the OPEN list, is marked as *visited*, and its “eligible neighbors” are added to the OPEN list. Whenever the goal is expanded, the algorithm is done, and the path leading to the expansion of g is returned as the final output path.

In path planning, eligible neighbors of a node are its *unvisited*, *collision-free* neighbors. That is, the costly collision detection operations are performed for the *unvisited neighbors of an expanded node* at every iteration. $A^\star$ has only one parallelization source: the eligible neighbors of an expanded node can be tested for collision in parallel. For example, with an 8-connected grid, up to eight collision detections can be parallelized (and typically far fewer). Other than this, $A^\star$ path planning is serial, as are most of its extensions and variants [37, 41]. The reason is the fact that the *optimality* of the algorithm depends on the *expansion order*, and (naive) parallelization can potentially disturb the order, sacrificing the optimality. This is a severe performance bottleneck given that modern mainstream computing systems support much more parallelism.

2.2.2 Patterns Exposed During Path Search. The green arrows in Figure 1 (left) show how the robot moves following the optimal path returned by $A^\star$. It first moves north (N), then *keeps* moving N for another two steps, then moves NE, then E, then *keeps* moving in the same direction for another six steps, and so on.

Observation III: The footprint of path exploration exhibits “cone-like” patterns.

Paths extracted in planning, exhibit regular, predictable patterns in state space: *connected straight-line segments* rather than frequent, irregular direction changes (green arrows in Figure 1 (left)). And, *exploration of those paths* manifests cone-like patterns: the footprint of traversing the graph mostly comprises *cones*, with few turns in direction, around each segment of the explored paths (purple cone in Figure 1 (left)). Figure 4 in §5.3 depicts the cones for a 2D benchmark.

The patterns arise partly because of the *geometric features* of path planning and partly because of *regular organization and structure* of real-world environments. Consider a collision-free 2D space in which a mobile robot tries to reach a destination from a start point. The shortest path between the two points is a straight line that connects them (a basic geometry principle). In such an environment, the robot will start to move in the direction of the goal and will *keep moving in the same direction* until it reaches the goal. In more general environments, the robot changes direction due to obstacles, but again otherwise keeps moving in the same direction. Moreover, the structure of many real-world environments encourages continuing in a given direction. Picture a self-driving car moving in a certain direction in a street bounded by buildings from the sides. Even in the presence of lane changes and overtaking, the vehicle will mostly move in a regular, straight direction (same as manual cars). As a result, the extracted paths in real-world environments mostly comprise connected straight-line segments, and the exploration of such paths (i.e., the graph search algorithm) exhibits cone-like patterns: each explored path is embraced by a cone.

Our argument is that *relying on the history of directions that have been taken so far, we can speculate on the path going forward.* Even in the short scenario of Figure 1 (left), two-thirds of the taken directions in the final path are the same as the preceding direction. Although the *set of explored nodes* in path planning is a superset of the final path, we show that *the history of directions can be effectively used to speculate on what nodes will be explored.*

Importantly, the cone-like patterns are spatial. That is, the consecutive expansions (in time order) do not necessarily exhibit any patterns. Search algorithms may explore more than one *growing tree (GT)* inside the graph (dashed lines in Figure 1 (right)), and their exploration can be temporally interleaved. There is not necessarily any pattern among the multiple GTs whose explorations are interleaved during path planning; the pattern is exhibited only in each GT independently.

Finally, the cone-like patterns are "conceptual" and are exhibited at the algorithm-level (i.e., *semantic*), and not necessarily at the underlying memory layout. Therefore, we argue that path planning exhibits *semantic spatial locality* and implement a spatial predictor in software, not in hardware where *semantic information is unavailable* (§5.7.2).

3 RUNAHEAD COLLISION DETECTION (RACOD)

This section presents the two main components of *RACOD*.

3.1 Collision Detection Accelerator (CODAcc)

CODAcc's task is computing the collision status of an *OBB*. There are two major challenges for a hardware design: (*i*) *OBB* size (in number of cells) is dependent on the robot's body shape and the planner's resolution unit, and hence, could be *different* from one robot to the next. (*ii*) Checking a large *OBB* entails checking many occupancy grid cells; given a *narrow* memory interface, naively loading memory addresses of the cells would result in serialization, possibly offsetting much of the benefits of hardware acceleration.

We address the first challenge by designing a *Hardware OBB* (*HOBB*) coupled with a *greedy scheduler*. *HOBB* is a *fixed-size* hardware unit (set of registers) on which the actual *OBB*, determined by the software, is loaded. *HOBB* uses $L = 10$, $W = 3$, and $H = 3$ registers to represent length, width, and height, respectively. When an *OBB* is larger than the *HOBB*, a greedy scheduler *partitions* the *OBB* on the *HOBB*, in *multiple* steps.

We address the second problem using a MapReduce-style hardware computation model: *all* memory addresses are generated in parallel (map), then they go through circuitry that *coalesces* requests to the same *cache blocks* (reduce). Ultimately, *a few* unique cache blocks are requested from memory. Below, we explain the details of these techniques, along with *CODAcc*'s other building blocks.

3.1.1 Processor-Accelerator Communication. Collision status is determined based on the occupancy grid information; thus, the accelerator should have access to it. A pointer to the beginning of the occupancy grid in memory and its size in different dimensions are sent to the accelerator via a queue-based configuration interface [18]. These parameters are used for generating occupancy grid memory addresses, and do not change *during* the planning stage. All other communications with the accelerator are performed via a single added instruction:

```
check_coll <dim> <cfg> <res>
```

`dim` is a 1-bit immediate value (can be a part of an opcode), indicating whether *OBB* is a rectangle (2D) or a cube (3D). `cfg` is a pointer to the *OBB* that should be tested for collision, and `res` is the memory location to which the collision status is written. As a communication convention, the *OBB* configuration, to which `<cfg>` points, is coded in a cacheline-aligned structure as shown in Table 1.

Table 1: *OBB* configuration encoding.

	origin	size	orientation
2D	(x_o, y_o)	(l, w)	$\sin\theta, \cos\theta$
3D	(x_o, y_o, z_o)	(l, w, h)	$\sin\alpha, \cos\alpha, \sin\beta, \cos\beta, \sin\gamma, \cos\gamma$

`origin` is the coordinates of *OBB*'s origin. `size` is *OBB*'s size in different dimensions. `orientation` is *OBB*'s orientation. In 2D, it is simply described by θ, the angle between the rectangle and x-axis. In 3D, it is defined by α, β, and γ, three angles each representing rotation around one axis (roll-pitch-yaw). Also, instead of sending the angles themselves, their sine and cosine are sent to the accelerator; this simplifies the accelerator design since it gets rid of including circuitry to implement trigonometric functions. All the arguments are 32-bit floating point numbers.

When the instruction is decoded, the core forwards it to the accelerator. The accelerator computes the collision result, as described below, and writes it to the memory location specified in the operand. Finally, the instruction is committed.

3.1.2 Data Path. Before explaining the data path, we explain one simple optimization we make to the occupancy grid's memory layout. Namely, we optimize for spatial locality by implementing the occupancy grid using `uint32_t` such that every grid cell occupies only one bit. This way, more nearby cells are captured in a single cache block, at a cost of having to do bit masks to extract the desired occupancy bit.

Figure 2 shows an overview of *CODAcc*. ① shows *CODAcc*'s address generation unit (AGU). AGU generates all to-be-checked cells' *locations* and then their memory addresses, storing them in ② *HOBB*. The cells' locations are generated using the configuration information (origin, size, and orientation). For example, for a 2D *OBB* (see Table 1), the location of its origin is (x_o, y_o), and the location of its top-right corner is $(x_o + l\cos\theta - w\sin\theta, y_o + l\sin\theta + w\cos\theta)$. After generating locations, the corresponding memory addresses are obtained according to memory layout semantics (row-major layout), using the occupancy grid memory address and its sizes in different dimensions (§3.1.1).

HOBB consists of a set of registers, each corresponding to a *specific cell* in the *OBB*. Every register keeps a key-value pair: the memory address of the location it corresponds to, and its occupancy status (collision or free; 1 bit). The figure is drawn to resemble the registers-*OBB* correspondences. Assuming zero angles, on the front pane, the bottom-left corner register represents (x_o, y_o, z_o) cell, and the top-right corner register represents $(x_o + L, y_o + W, z_o)$ cell. Every register holds the corresponding cell's memory address and its occupancy status. Also, as we will describe shortly, the *same HOBB* is used for both 2D and 3D *OBB*s but with *different circuitry.*

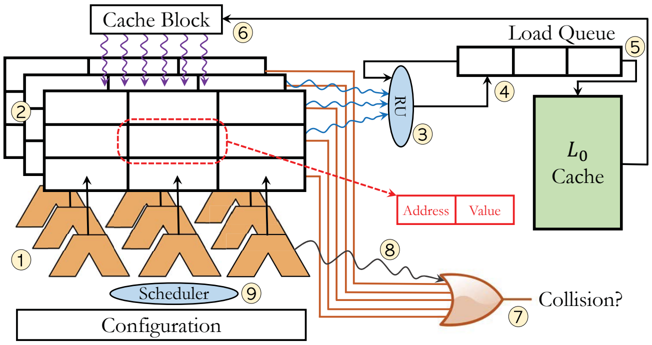

Figure 2: Hardware realization of *CODAcc*.

The generated addresses pass through a ③ reduction unit (RU), and then the requests for distinct *cache blocks* enter a ④ load queue (LQ). When the LQ is empty, the cache block request of the first[1] non-empty register enters the LQ. Then, RU performs an *associative search (i.e., parallel)* to find which registers need addresses that fall into the cache block at the head of the LQ, marks them as *pending*, and enqueues the cache block request of the first non-empty, non-pending register into the LQ. This repetitive process is stalled if the LQ becomes full, and finishes when no non-pending register remains.[2]

Noteworthy, due to the high spatial locality (§2.1), this process repeats only a few times. One cache block alone incorporates 512 bits (cells), while all registers together request 90 bits. As such, a few cache blocks serve all requests, and an 8-entry LQ is rarely filled up in practice.

LQ entries are constantly ⑤ dequeued and sent to the memory hierarchy. Upon a cache block arrival ⑥, registers whose addresses fall into that cache block take their value. The 1-bit values are placed into the registers, and are ⑦ ORed to produce the collision detection output. This process continues until either (*i*) the outcome of the OR gate rises anytime during the check, or (*ii*) the entire *OBB* has been checked.

Note that the entire *load-to-OR* path (④-⑤-⑥-⑦) works in a pipelined manner. I.e., the accelerator does *not* wait for all the loads before ORing them; a cell can raise the output of the OR gate as soon as its value is received from memory. This massive parallelism provided by pipelining/ORing in hardware, rather than checking one-by-one in software, is the major contributor to *CODAcc*'s performance improvement.

A corner case arises when an *OBB* extends outside the environment boundaries—it is an *invalid* configuration. When any of the memory addresses falls outside the occupancy grid's address range, the output is ⑧ short-circuited. Finally, when the collision result is computed, the registers are cleared.

When an *OBB* is smaller than the *HOBB*, some of its registers are left unused. To avoid having separate valid bits in registers and setting/resetting them, the unused registers in every dimension take the address of the last register in that dimension. This way, in fact, we include some states multiple times in our collision computation, but note that doing so does not affect the outcome of computation (bitwise OR). When an *OBB* is larger than the *HOBB*, ⑨ the scheduler *partitions* it, performing its collision detection in multiple serial steps.

To design a simple yet efficient scheduler, we deem the partitioning as an optimization problem whose goal is *to maximize cache hits* across multiple steps. We use this greedy algorithm: first, fully evaluate the x dimension, which will be done in $\lceil \frac{l}{L} \rceil$ steps, l being the length of the *OBB*; then complete the y dimension; and finally, complete the z dimension (if 3D). We prioritize x over y (and y over z) to leverage the row-major layout of multi-dimensional arrays in memory, for cases when the *OBB* is axis-aligned or nearly so. This is also in part a reason why we chose a large L for the *HOBB*.

Finally, although the location of cells in a 2D *OBB* can be computed by the same circuit used for 3D (by zeroing the third dimension), we dedicate separate circuits to the AGU and scheduler of 2D and 3D computations. This has two benefits: (*i*) a 2D *OBB* can be computed faster, and (*ii*) when a 2D *OBB* is large, the scheduler can dispatch parts of it on idle z registers, computing the collision status in fewer steps.

3.1.3 L_0 Cache. We provision the accelerator with a 256-byte (2048-bit) L_0 cache. This L_0 caches the data requested by the AGU. The L_0 efficiently filters the majority of requests not only by exploiting spatial locality, but also temporal locality: subsequent collision checks have a large amount of overlap.

3.1.4 System Integration. A processor can be integrated with multiple instances of the accelerator. When so, like other functional units (adders, multipliers), the core's scheduler is responsible for dispatching different `check_coll` instructions to *CODAcc* units. Also, every *CODAcc* unit has its own L_0 cache; all L_0 caches are backed by the core's L_1 cache and forward misses to its interface (a 16-entry queue).

To keep the entire system coherent, blocks cached in L_0 are marked in the processor's L_1 cache (1-bit extension); whenever a marked block is evicted from L_1 or invalidated or written, the block is invalidated in L_0. The cache extension overhead is 128 bytes per core (not per accelerator). Also, L_0 is virtually indexed, physically tagged. A TLB with a couple of entries is sufficient to translate nearly all accesses.

3.2 Run-Ahead State Exploration (RASExp)

3.2.1 Baseline Algorithm and Extension. As we show later, *CODAcc* is so low overhead that we can afford many instances of it. However, the limited parallelism of the search algorithm (§2.2.1) limits the benefits of having many *CODAcc*s. *RASExp* is a technique to increase parallelism: at every step, it *predicts* likely-to-be-explored next states, *speculatively* performs their collision checks in parallel with those of the current state, and *memoizes* the collision status for potential later usage. The key insight of *RASExp* is that expansions with the search algorithm exhibit cone-like patterns (§2.2.2).

[1] The order among registers is hardwired; e.g., `reg0` precedes `reg1`.

[2] The RU's reduction mechanism is different from that of caches' miss status holding registers (MSHRs). MSHRs handle requests one-by-one: the first request triggers a cache miss, the block address is stored in MSHR, and subsequent accesses to the outstanding cache block are served one-by-one. The RU uses a different approach: all requests are reduced *at the source* and *in parallel*. Also, the RU is different from recent GPUs' address coalescers [15]; the RU supports bit-granular, irregular (oriented) address coalescing while most GPUs coalesce *regular* addresses at a word granularity.

```
01   exp_node = OPEN.pop()
02   // evaluate the expanded node ...

03   for n in exp_node->neighbors():
04       if !visited[n] and collision_status[n] == UNKNOWN:
05           collision_status[n] = PENDING
06           thread {collision_status[n] = check_collision(n)}

07   if any_outstanding_thread:
08       ll_counter = MAX_DEPTH
09       pred_dir = get_dir(exp_node, exp_node->parent)
10       pred_n = exp_node

11       while freeContexts > 0:
12           pred_n = pred_n->neighbors()[pred_dir]
13           for n in pred_n->neighbors():
14               if !visited[n] and collision_status[n] == UNKNOWN:
15                   thread {collision_status[n] = check_collision(n)}
16                   if freeContexts == 0: break
17           if --ll_counter == 0: break

18   all_threads.join()

19   for n in exp_node->neighbors():
20       if !visited[n] and collision_status[n] == FREE
21           // evaluate n and push to OPEN
```

RASExp's Extension

Algorithm 1: Baseline $A^{\star}$ and *RASExp*'s extension.

RASExp's prediction mechanism is very simple: *the path will grow in the same direction as it grew in the last step.* Therefore, whenever a node is expanded, *RASExp* finds out the direction that led to the expansion, and predicts that *the path will grow in the same direction.*

Algorithm 1 shows the main iteration of both the baseline $A^{\star}$ and *RASExp*'s extension. The pseudo-code depicts a multithreaded baseline $A^{\star}$, as well as a multithreaded *RASExp*. With a *CODAcc*-rich processor, each thread call gets replaced by a check_coll instruction.

In line 01, the node with the minimum f is expanded (§2.2.1). Then some basic operations of $A^{\star}$ are performed (e.g., mark *visited*). Starting at line 03, the planner looks for *eligible neighbors*. For every unvisited neighbor, if its collision status is unknown, a thread computes its collision status.

In the *absence* of *RASExp*, $A^{\star}$ waits for threads to join (line 18). It then evaluates unvisited, collision-free neighbors and (potentially) adds them to the OPEN list (lines 19–21).

RASExp (lines 07–17) tries to *speculatively overlap* future nodes' collision operations with outstanding collision checks. First off, it is done only if there are outstanding collision checks (line 07); i.e., *RASExp* does *not* stall the main execution thread for speculative operations.

RASExp extracts the direction that led to current expansion (line 09), and predicts the path will grow in the same direction (lines 10 and 12). Then, as long as a free context (thread or *CODAcc*) exists, *RASExp runs ahead* and offloads a collision detection to it (lines 11–17). The computation of these collision checks (*speculative*) is *overlapped* with the computation of outstanding ones (*demand*).

Finally, to avoid *livelock*, *RASExp* uses an ll_counter, initialized with MAX_DEPTH (line 08), and decrements it after every run-ahead, and halts the process if it expires (line 17). In this paper, the default MAX_DEPTH is 8. Hence, *RASExp* can run up to eight vertices ahead, and perform the collision checks for each of their neighbors.

3.2.2 *Discussion & Optimizations.*

RASExp is a radically different parallelization approach. Hence, it opens up new opportunities and poses different design questions.

Sophisticated Predictors: *RASExp*'s prediction mechanism is simple: *the last direction in the GT (§2.2.2) will repeat.* This can be replaced by a sophisticated predictor to capture more complex patterns (e.g., zigzag patterns). Our current workloads do not justify such sophisticated predictors (§5.7.1); however, we believe that complex predictors could be effective in other applications of $A^{\star}$ (e.g., Protein design [51], natural language processing [27]) or in other graph search algorithms.

Hardware Prediction: One might wonder why the prediction is not made in hardware. The answer is: while the patterns are regular semantically, they are not so in hardware. First, as discussed in §2.2.2, the expansion of GTs inside the graph can be interleaved; a hardware predictor could be bewildered by this issue alone. Second, the spatial patterns are conceptual, and the trees do not necessarily exhibit those patterns in the memory layout. For example, consider a tree growing in a diagonal direction: while conceptually the tree is growing in a straight direction, the memory addresses can be mapped to distant locations in the memory layout.

4 METHODOLOGY

We evaluate our hardware accelerator alone and in conjunction with our algorithm extension, using simulation, in §5. We also evaluate a software-only implementation of our algorithm extension on commodity hardware (CPU and GPU) in §6.

We write CPU applications in C++17 and compile them using GCC 11. We develop GPU applications using CUDA 11 and compile them with NVCC. We compile the codes with the maximum optimization level (-O3).

We synthesize the accelerator in TSMC's 45-nm ASIC flow, using the Synopsys Design Compiler. We perform simulations using ZSim [39], and model a processor after the Intel Core i3-8109U [5]—a state-of-the-art robotic processor deployed in LoCoBot [2]. We simulate all programs to completion.

We conduct our software-only evaluations using a 32-core Intel E5-2670 CPU [1] and an NVidia GeForce GTX 1060 GPU [4]. We use Ubuntu 18.04 with Linux Kernel 4.15 as our operating system.

5 EVALUATION

5.1 Accelerator's Specifications

Table 2 shows the design parameters of *CODAcc* in 45-nm technology. The 'Power' represents the total power at maximum activity estimated by the synthesis tool.

Table 2: Design parameters of *CODAcc*.

Component	Cycles (@3 GHz)	Area (mm^2)	Power (mW)
Logic+Registers	5	0.019	12.1
L_0 Cache	1	0.004	0.17
Total	-	0.023	12.27

CODAcc has a simple, small structure: 10 × 3 × 3 registers plus simple logic to implement operations like addition, multiplication, and comparison. The accelerator takes $0.023mm^2$ and consumes $12.27mW$. As a point of comparison, in 40 nm technology, a comparable Intel processor's die-size is $276mm^2$, and its power consumption is $94W$; a single core of the processor alone occupies $25mm^2$ of silicon area and consumes $11W$ power [32]. Even tiny ARM Cortex-A15 cores in 40 nm technology occupy $4.5mm^2$ and consume $1W$ [32].

Due to its low overhead, we can integrate tens of *CODAccs* with the processor. The area overhead of thirty-two *CODAccs* altogether plus the cache extension (§3.1.4) is less than $0.73mm^2$ (3% of a core's area and 0.3% of the die-size). Also, thirty-two accelerators consume less than $393mW$ at full load (3.5% of a core's power and 0.5% of chip power).

5.2 Mobile Robot Navigating in 2D

First, we evaluate a mobile robot navigating in 2D environments. The program resembles a self-driving car navigating in a city. We use snapshots of four cities available in Moving AI [42]. Figure 3 (top) shows snapshots of the environments, and Figure 3 (bottom) shows *RACOD*'s speedup on them, varying the number of accelerators.

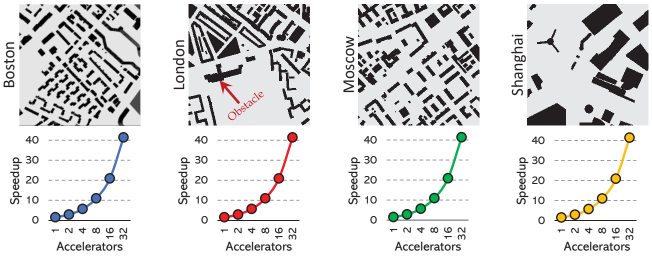

Figure 3: 2D navigation in the wild.

For every map, we choose 100 random start/goal points. The graph is 8-connected, and a multithreaded $A^\star$ is the baseline algorithm. In the baseline implementation, 67.3% of the entire path planning time is spent in collision detection.

One *CODAcc* alone improves performance by 1.49×. This is the speedup of pure hardware acceleration (no *RASExp*), obtained from parallelizing *individual* collision checks. *RASExp* further enhances performance by parallelizing *different* collision checks. *RASExp* greatly scales up the parallelism, achieving 41.4× speedup with 32 *CODAccs*.

Interestingly, we observe similar *normalized* speedups with different maps. This mainly emanates from the high prediction coverage/accuracy of *RASExp* (see §5.7.1), which results in predicting *enough correct nodes* in all the environments and thereby effectively keeping all the accelerators utilized (see §5.8). As a result, *RACOD* brings *linear* speedup proportionate to *speculation runahead* across all the maps.

5.3 Exploration Footprint

Figure 4 shows an approximation of all the nodes explored during the search (not just the final path) in one planning scenario, where we have zoomed-in on part of the full map. As shown, the majority of speculations are accurate (green), with only a few misspeculations (red) typically happening on the fringe of heavily-explored areas.

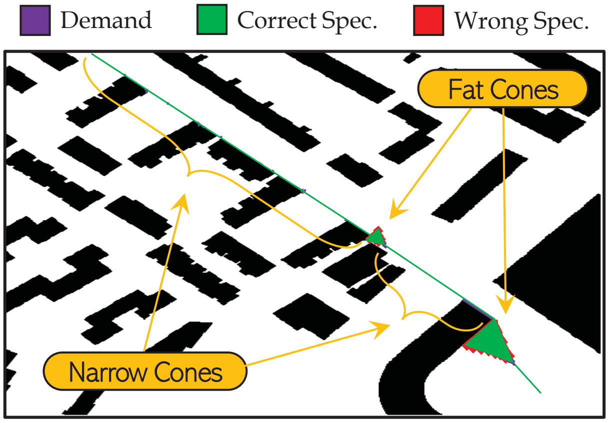

Figure 4: Cone-like patterns in a Boston snapshot, with a runahead of 32.

The figure also visualizes cone-like patterns: *fat cones* (when the planner struggles to find an efficient path through a cluttered area), and *narrow cones* (when the planner keeps exploring an uncluttered, straight-line path towards the goal).

By running ahead of a path and proactively evaluating *neighbors* of prospective nodes, *RASExp* effectively captures the exploration patterns (quantitative results in §5.7).

5.4 Mobile Robot Navigating in 3D

Next, we evaluate a mobile robot navigating in a 3D environment. The program resembles an unmanned aerial vehicle (UAV), a.k.a. drone, navigating in an outdoor environment. We use the 'Freiburg campus' map available in the OctoMap 3D scan dataset [47] as our environment. The resolution of the map is 0.2^m. Figure 5 shows the environment (left) and *RACOD*'s performance improvement with different numbers of accelerators (right).

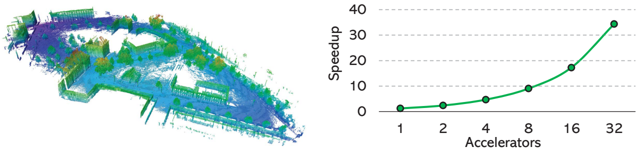

Figure 5: A map of the environment and the performance improvement.

We choose 10 random start/goal points. The UAV can move back and forth in all three dimensions. On average, the baseline spends 54% of the entire path planning time performing collision detections.

One *CODAcc* alone accelerates planning by 1.24×. With *RASExp*, *RACOD* substantially scales up the parallelism, providing 34.3× speedup with 32 *CODAccs*.

5.5 Robotic Arm Operating in 3D

As a proof of concept for our accelerator's applicability to a wider domain, we further study a robotic arm planning application: a stationary robotic arm with multiple *degrees-of-freedom (DoF)* operating in a 3D environment.

Robotic arm planning has as many dimensions as its DoF. High-dimensional planning is performed by *sampling* the configuration space. *Rapidly-exploring Random Trees (RRT)* [29] is a widely-used robotic arm planning algorithm. The main advantage of *RRT* over former methods like *PRM* [25] is the ability to work in *arbitrary* environments, which is also a major design consideration of our accelerator.

RRT extends a tree (*not a more general graph*, as in mobile robots) from the start point by drawing random samples; the tree is extended towards *collision-free* samples until reaching the goal.

We model a robotic arm based on an in-house 5-DoF LoCoBot [2], operating in the environment shown in Figure 6 (left), planned by a state-of-the-art parallel *RRT* [28]. The arm moves from $s = (-80°, 0°, 0°, 0°, 0°)$ to $g = (0°, 60°, -75°, -75°, 0°)$. With this (s, g) pair, the arm traverses a long trajectory with different types of movement (translation and rotation), forming various configurations for collision detection. On average, the baseline spends 80.5% of the planning time in collision detection. The robot is bounded by the *OBB*s shown in Figure 6 (middle). Figure 6 (right) plots the speedup with 1–4 accelerators.

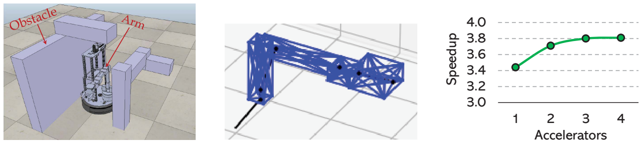

Figure 6: The modeled robot and environment (left), LoCoBot's *OBB*s (middle), and the performance improvement of hardware acceleration (right).

Note that *RASExp* is not applicable nor needed in *RRT*. Since *RRT* creates a tree, not a graph, there is no need to search the structure to find a path. The path is simply *extracted* by traversing each node's `parent` pointers from the goal to the start. Nevertheless, multiple *CODAcc*s can enable parallel collision status computation of different *links* of the arm.

One *CODAcc* improves the execution time by 3.4×. With increasing the number of *CODAcc*s up to the number of *OBB*s, the performance increases slightly, up to 3.8×.

5.6 CPU-Accelerator Communication Latency

When an accelerator is not tightly integrated with the CPU, the communication latency can go up and hurt the performance, sometimes rendering the accelerator harmful [18].

In this section, we evaluate three communication latency numbers: 1 cycle (tightly integrated, default), 10 cycles (co-processor, system-on-chip), and 100 cycles (off-chip). In the two latter cases, we assume the communications are explicitly established by the programmer: the programmer copies all the configurations that should be tested for collision into a buffer, triggers the accelerator, and gathers all the results at once when they are ready.[3] Also, the communications have blocking semantics, meaning the processor waits for the operation to finish before proceeding. Figure 7 shows the results. With '1 *CODAcc*,' all robots have only one accelerator, and in '32/4 *CODAcc*s,' the mobile robot has 32 accelerators and the arm has 4.

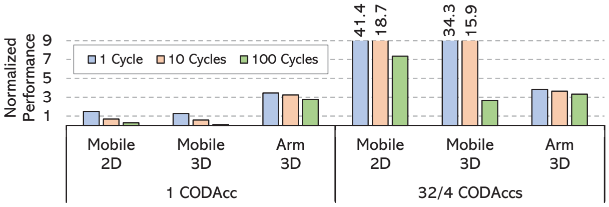

Figure 7: Speedup sensitivity to CPU-accelerator communication latency.

When the system has only one accelerator, the performance is very sensitive to communication latency. When the number of accelerators (runahead/parallelism) increases, the communication overhead gets amortized, especially in mobile robots in which 32 *CODAcc*s are deployed.

5.7 Prediction Coverage and Accuracy

5.7.1 *Semantic Predictor.* Recall that *RASExp* uses semantic information and is implemented in software (§3.2). Figure 8 (top) shows its prediction accuracy (bars) and coverage (dots) with different runaheads (R). Prediction accuracy is the percentage of predictions whose computation result is eventually used by the planning algorithm. Prediction coverage is the percentage of speculated collision checks that must otherwise (i.e., without *RASExp*) be done non-speculatively.

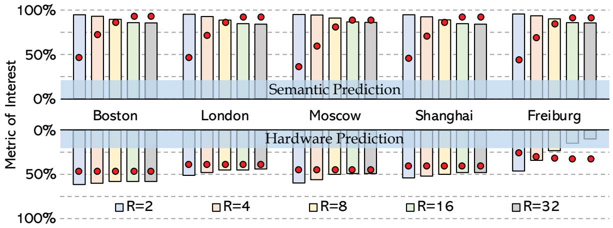

Figure 8: Prediction accuracy/coverage (bars/dots) with different runaheads.

With a runahead of two, 95.1% of predictions are accurate, substantiating our observation on conceptual, *semantic spatial locality* in node expansions. Also, the prediction coverage is 43.4%. With increasing the runahead, *RASExp* becomes more aggressive, offering higher coverage and slightly lower accuracy. With a runahead of thirty-two, *RASExp*'s coverage reaches 90.9%, while offering more than 85.1% accuracy.

[3] Copying latency is assumed to be captured in communication latency.

Incorrect predictions result in energy wastage, as the collision status is computed but never used. Nonetheless, because the prediction accuracy is so high, and the *CODAcc*s, on which misspeculations run, are so low power, the energy wastage of *RASExp* is quite negligible ($\ll$0.01% of chip power).

5.7.2 Hardware Predictor. Next, we study the effectiveness of *RASExp* implemented in hardware. Since child-parent relations are lost, *RASExp*'s *simple* prediction method cannot be used in hardware. In Figure 8 (bottom), we study *RASExp* with a state-of-the-art hardware predictor.

We *repurpose VLDP* [40], a state-of-the-art pattern prefetcher, to predict future states in planning. We make several changes to *VLDP*'s design: (*i*) We use infinite-size metadata tables. (*ii*) We trigger the predictor *only* upon collision detection accesses; this lets the predictor observe one clear-cut access stream, rather than many interleaved streams. (*iii*) The predictor operates on *virtual addresses.* (*iv*) The predictions are stored in infinite storage (prefetch buffer in prefetching terminology). All four changes are in favor of the prediction accuracy and coverage of hardware prediction.

The coverage and accuracy of semantic prediction are significantly higher than those of hardware prediction: 2.1× coverage and 2× accuracy on average. This is particularly true in the drone application where the addition of a third dimension completely bewilders the hardware predictor. The results reinforce the importance of exploiting semantic information that is difficult to extract in hardware.

Noteworthy, footprint-based spatial pattern predictors [23] could *not* be repurposed for this experiment. Those predictors collect patterns when the tracked region is *evicted* from the cache, while the notion of cache does not exist in this problem. Also, temporal prediction [13] would be meaningless in this context since collision detection sequences never repeat: there is at most one collision check per state.

5.8 Division of Labor

The bars in Figure 9 show the average number of useful collision checks per node expansion. *Demand* represents the collision checks performed by the baseline algorithm, and *speculative* represents those issued by *RASExp.* With increasing runahead, the contribution of the speculative computations goes up: more on-the-critical-path collision checks are performed speculatively ahead of time, and as a result, the planning is *less stalled* on every expansion.

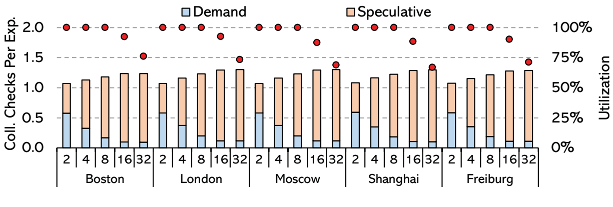

Figure 9: Division-of-labor, varying the number of accelerators.

The solid dots in Figure 9 show the utilization ratio of the accelerators (threads) in *non-idle* expansions, i.e., expansions in which at least one collision detection is performed. With a handful of accelerators (2–8), the utilization ratio is nearly 100%, showing that the amount of parallelism with the baseline algorithm plus the additional parallelism provided by *RASExp* is high enough to always keep the accelerators busy. With more accelerators, the utilization ratio decreases, mainly because of the livelock-avoidance mechanism (§3.2.1).

5.9 Weighted $A^\star$ & Different Heuristics

$A^\star$ is guaranteed to find the shortest path (*optimal*), with the minimum number of expansions (*optimally efficient* [16]). However, not all applications require finding the shortest path: some applications favor a suboptimal path to an optimal path, if finding the suboptimal path is significantly faster.

Weighted $A^\star$ *(*$WA^\star$*)* [38] is the most popular satisfying algorithm for heuristic search in various domains [46]. $WA^\star$ *inflates* the heuristic by a factor of $\varepsilon > 1$. That is, $WA^\star$ expands nodes in the order of $f(v) = g(v) + \varepsilon \times h(v)$. This way, the search is biased towards the nodes that are closer to the goal, resulting in faster expansion of the goal. On the flip side, the final path cost could become ε times higher than the shortest path cost.

Moreover, prior work proposes various heuristics $h(v)$. So far, we used *Euclidean distance* as our heuristic. In this section, we reevaluate the experiments of §5.2 with two other popular 2D heuristics: *Manhattan distance* and *non-uniform diagonal distance* [10]. We also evaluate the Dijkstra search algorithm, which does not use any heuristics.

Figure 10 shows speedup (bars) and prediction coverage (dots) with different heuristics and weights, averaged across all workloads. The speedup is the performance of every method with *RACOD* normalized to that without *RACOD*. All evaluations are done with 32 threads/accelerators.

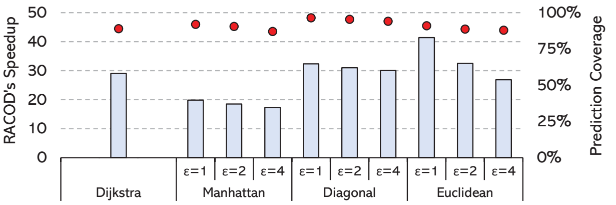

Figure 10: *RACOD*'s effectiveness with $WA^\star$ and different heuristics.

RACOD consistently brings significant speedup for all methods, showing its applicability to a wide range of algorithms and heuristics. With increasing the weight (ε), the improvement (slightly) drops, particularly because of the reduced prediction coverage, which itself is caused by the fact that fewer nodes are expanded with larger ε values.

Not shown in the figure, inflating the heuristic by a factor of 2/4 accelerates planning by 1.6×–2.2×/2×–3.8×. Also, Dijkstra is on average 25× slower than $A^\star$, and the performance of different heuristics is within 1.2×–5.3× of each other.

5.10 L_0 Cache Configuration

Figure 11 shows L_0 hit ratios with different sizes. As shown, a 256 B cache is sufficient to filter the majority of requests.

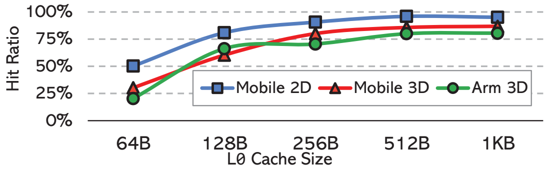

Figure 11: L_0 cache hit ratio with varying L_0 size.

Note that the major benefit of L_0 is lifting *bandwidth* pressure from the core's L_1 cache. Latency is of less concern because: (*i*) all the requests are generated in parallel and their latency, in case missed in L_0, gets well overlapped, and (*ii*) L_0 misses are often served by the L_1, whose latency is not high.

5.11 Controlling Prediction Aggressiveness

RASExp's prediction mechanism is aggressive: it is *always* triggered—this gives the highest coverage/performance in the evaluated benchmark environments. But in some rocky environments with frequent, irregular direction changes, or with platforms with severe power constraints, it might be beneficial to *throttle* the predictor in order to avoid making numerous wrong predictions. To reduce the aggressiveness, *RASExp* employs this algorithm: the predictor is triggered only if the path leading to the expanded node was *stable* for at least s steps. E.g., with $s = 3$, the predictor is triggered only if a node's expansion direction is the same as the expansion direction of its parent and its parent's parent.

We create synthetic city-resembling maps (§5.2), in which, with a probability of 10%–70%, we inject *random* obstacles to an initially free space. Figure 12 shows how the prediction accuracy (left) and coverage (right) of *RASExp* vary. In this experiment, the runahead is 32.

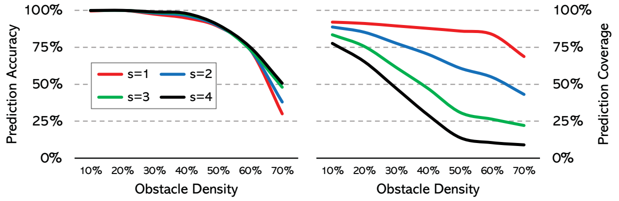

Figure 12: Prediction throttling impact with different obstacles densities, varying the predictor trigger threshold s.

RASExp's throttling mechanism is quite effective: with $s = 4$, it successfully harnesses the predictor's aggressiveness such that even in an environment with 70% *random obstacles*, the accuracy is still above 50%. On the flip side, the coverage drops as a result of reduced prediction opportunities.

Another important takeaway of this experiment is the significant difference in prediction accuracy/coverage numbers between synthetic and realistic environments (e.g., with $s = 1$, 39%/68% accuracy/coverage for the 70%-random environment versus 85%/90% for the benchmarks). As discussed in §2.2.2, the organization of real environments is not so irregular that it would destroy patterns that emanate from geometric features of path planning.

6 RUNAHEAD MULTITHREADING

Parallelizing path search algorithms like Dijkstra, $A^\star$, and $WA^\star$ is not straightforward, because the *optimality* (or ε-optimality) of the algorithm depends on the *expansion order*. Naively parallelizing *different expansions* can potentially disturb the correct expansion order and greatly sacrifice the optimality. Prior work [12, 19, 22, 26, 37, 43, 44] proposes methods for *safe* parallelization of expansions. For example, *PA***SE* [37] parallelizes the expansion of *independent states*: if the expansion of s cannot lead to a *shorter* path to s', and vice-versa, they are independent and their expansions can be reordered (safely parallelized).

RASExp is a fundamentally different approach. It does *not* change the expansion order and is faithful to the underlying algorithm's execution flow. It *predicts* future expansions, *pre-computes* their collision status, and *memoizes* them for when the actual expansion takes place: no expansion order is changed. In fact, *RASExp* is a *speculation* technique, *necessarily* done at the algorithm level. Speculation never changes a program's behavior but accelerates it.

Figure 13-(a, b) show the speedup of (*i*) $A^\star$ with *Baseline Multithreading (BM)* (on expansion, all the node's eligible neighbors are evaluated in parallel), (*ii*) *PA***SE* [37], and (*iii*) $A^\star$ with *RASExp*, over the corresponding single-threaded implementation on CPU and GPU platforms. In this experiment, we consider the average of the mobile robot workloads.

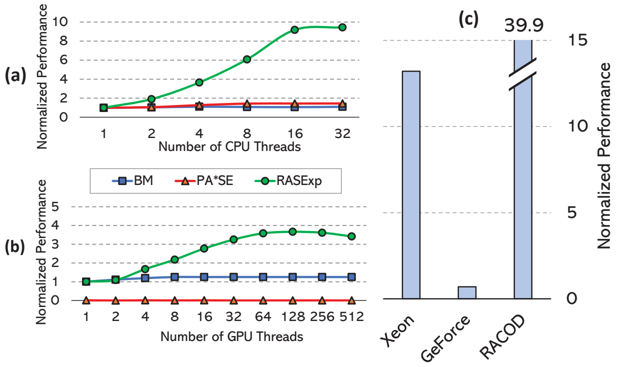

Figure 13: Performance comparison of different platforms and configurations. All but RACOD are commodity hardware.

On the Xeon CPU, *BM* has limited speedup: 9% speedup with 32 threads. The reason is the severely limited parallelism of the baseline path search algorithm (§2.2.1). *PA***SE* fails to significantly improve performance: 45% speedup with 32 threads. *PA***SE* suffers from two fundamental issues: (*i*) There are not enough independent states when running the planning algorithm to fully utilize all available cores (the evaluations in [37] were only up to 8 cores). (*ii*) The overhead of finding independent states is high; this issue is acknowledged in [37], but because the authors' evaluations were on a large PR2 robot [14] with too high resolution, the cost was amortized to an extent.

By accurately predicting future expansions, *RASExp* significantly improves performance, decisively outperforming the other two.

With 32 threads, *RASExp* improves performance by 9.44×/8.59×/6.5× over single-threaded/*BM*/*PA***SE*.

On the GeForce GPU, more threads are available. As such, we relax the livelock-avoidance mechanism, and set `MAX_DEPTH=64`, which results in discovering (theoretically) up to $64 \times 8 = 512$ nodes during speculation.

As the GPU results show, *BM* exhibits a similar behavior for the reasons outlined above. The performance improvement of *RASExp* is not as significant as on the CPU. There are two major reasons: (*i*) A larger portion of the execution time is spent in the *serial* part of the algorithm, because it is significantly GPU-averse (e.g., giga-scale data structures, pointer chasing during insertions). (*ii*) Collision detection on multiple GPU threads generates a large number of *branch divergences*, since threads check different parts of the environment whose occupancy status could be different. Also, we observe performance degradation after 128 threads; the prediction accuracy significantly drops (17.3% with a runahead of 256) and the speculation overhead (lines 11–17 in Algorithm 1) grows.

PA**SE* is totally inefficient on the GPU; its execution time is an order of magnitude longer than the single-threaded baseline. The main reasons are: (*i*) Checking independence conditions, which is done serially for a large number of nodes, take a huge amount of time. (*ii*) *PA***SE*'s need for larger and more numerous data structures (e.g., `set` to track *being-expanded* nodes) makes the method more GPU-averse than the baseline.

Figure 13-(c) compares the performance of the CPU and GPU platforms with that of *RACOD*. The CPU and GPU software enjoy *RASExp* with runaheads of 32 and 128, respectively; these configurations offer the maximum performance that we can achieve on *high-end* CPUs and GPUs, without hardware modifications. Our proposal, *RACOD*, offloads the collision detection operations on 32 *CODAcc*s and runs the rest of the planning on a *low-end* Intel Core i3-8109U Processor [5], a typical processor used in modern robotic systems like the modeled LoCoBot [2]. The performance metric is the wall clock time of the execution, and is normalized to a software-only baseline (no *CODAcc*, multithreaded but no *RASExp*), executed on an Intel Core i3-8109U [5].

The GPU platform is clearly unfit for mobile robot path planning, because the software is significantly GPU-averse. The CPU platform with high-performance cores is able to considerably improve the performance over the Intel Core i3-8109 baseline (13.2× on average), given that the algorithm is equipped with *RASExp*. However, this performance comes at the cost of powering four processors with 115W TDP, operating within NUMA/sockets. *RACOD*, using a low-end processor with 28W TDP and 32 *CODAcc*s that in total consume $< 0.5W$, improves performance by 39.9×, outperforming the other platforms and underscoring the importance of hardware acceleration.

7 RELATED WORK

7.1 Hardware Acceleration for Path Planning

In the context of path planning, a few proposals design accelerators for some narrow domains. Particularly, Murray et al. [35] devise an FPGA-based *PRM* planner for a stationary robotic arm functioning in their laboratory. They test various planning scenarios offline and find that with only 1024 movements (called *edges*), > 98% of planning scenarios in their environment can be accomplished. Then they find the environment points from which those movements cross and store the entire information on an FPGA. During operations, they perform collision checking for every movement by simply checking the occupancy of the stored points in the environment. The main limitation of this approach is its tight integration with the environment: if objects in the environment change, the offline process, which could take hours, needs to be repeated from scratch.

Dadu-P [31] uses a similar approach but admits *some* obstacle movement rather than assuming a fixed environment. *Dadu-P* uses more edges, some of which may cross obstacles in the environment that are likely to move (e.g., a wall is not supposed to move, but a chair is). During planning, the edges are tested for collision, based on their latest organization.

While both approaches accelerate the planning, neither are scalable. Storing a large number of edges, in the evaluated $3.5^{cm} - 7.5^{cm}$ resolutions, occupies an entire Stratix V FPGA in [35] or costs 768 Kb SRAM storage in [31]. In larger environments, or in environments at the same scale but with finer-resolution planning, much more movements would be required to cover the majority of planning scenarios, demanding significantly larger storage.

More importantly, both approaches make restrictive assumptions about the environment (the reachability of certain points through certain paths, (most) obstacles never move) that do not necessarily hold in many applications, *especially in mobile robots where the environment changes constantly and unpredictably*.

7.2 Path Planning & Parallelism

A myriad of proposals in the robotics community aim at tuning efficient algorithms and heuristics to accelerate path planning. Still, state-of-the-art CPU and GPU proposals are *not* fast enough to be considered real-time in many applications [45]. We believe our design is applicable to a wide range of prior proposals, because it relies on fundamental characteristics of path planning like spatial locality in collision detection and patterns in path search.

Outside of the context of path planning, several pieces of prior work have proposed to exploit *ordered parallelism* [50] by speculatively executing different (atomic) tasks of task-based programs: codes with myriads of *short* tasks that should be executed based on the *timestamps* specified by the programmer. Thread-Level Speculation (TLS) and Hardware Transactional Memory (HTM) [6, 8, 21, 24, 48] execute different tasks speculatively, committing successful speculations and *aborting* wrong ones. Wrong speculations (i.e., parallel execution of *dependent* tasks) are usually detected by relying on the cache coherence protocol, and the conflicting tasks are re-executed from scratch.

RASExp is a fundamentally different approach: it exploits application semantics to *predict* future paths, rather than executing and then monitoring shared-data accesses. Also, *RASExp* parallelizes computationally-intensive collision detections, while TLS and HTM parallelize short, atomic tasks. Further, *RASExp does* accelerate dependent tasks, in TLS/HTM terminology, as it proactively evaluates an expanded node's (grand)children, while TLS/HTM methods avoid doing so. Finally, most TLS and HTM approaches apply heavy microarchitectural modifications to detect conflicts and queue/manage/recover tasks, while *RASExp* is a semantic technique and does not cause conflicts.

8 CONCLUSION

Deploying robots in the wild requires the development of real-time computational solutions. In this work, we study path planning, a core module in autonomous robots, and propose an algorithm/hardware co-design to substantially improve mobile robot path planning performance.

We exploit architectural-level computation characteristics of mobile robot path planning, as well as its high-level semantic features, to massively parallelize the kernel. Specifically, (*i*) we architect cheap hardware accelerators to exploit fine-grained parallelism and spatial locality in costly collision detection operations, and (*ii*) we enable proactive exploration by semantically predicting directions along the search path. Our future work is focusing on other application kernels in real-time robotics [9], exploring similar architectural and semantic acceleration techniques.

ACKNOWLEDGMENTS

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.

REFERENCES

[1] 2012. Intel Xeon Processor E5-2670. https://ark.intel.com/content/www/us/en/ark/products/64595/.

[2] 2012. LoCoBot: An Open Source Low Cost Robot. http://www.locobot.org/.

[3] 2015. Arduino Ant Hexapod Robot. https://antdroid.grigri.cloud/.

[4] 2016. GeForce GTX 1060. https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1060/.

[5] 2018. Intel Core I3-8109U Processor. https://ark.intel.com/content/www/us/en/ark/products/135936/.

[6] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[7] Ron Alterovitz, Sven Koenig, and Maxim Likhachev. 2016. Robot Planning in the Real World: Research Challenges and Opportunities. *AI Magazine* (2016).

[8] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. 2020. Perspective: A Sensible Approach to Speculative Automatic Parallelization. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[9] Mohammad Bakhshalipour, Maxim Likhachev, and Phillip B. Gibbons. 2022. RTRBench: A Benchmark Suite for Real-Time Robotics. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://cmu-roboarch.github.io/rtrbench.

[10] Sven Behnke. 2003. Local Multiresolution Path Planning. In *Robot Soccer World Cup*.

[11] Joshua Bialkowski, Sertac Karaman, and Emilio Frazzoli. 2011. Massively Parallelizing the RRT and the RRT*. In *International Conference on Intelligent Robots and Systems (IROS)*.

[12] Ethan Burns, Seth Lemons, Wheeler Ruml, and Rong Zhou. 2010. Best-First Heuristic Search for Multicore Machines. *Journal of Artificial Intelligence Research* (2010).

[13] Trishul M. Chilimbi and Martin Hirzel. 2002. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*.

[14] Steve Cousins. 2010. ROS on the PR2 [ROS Topics]. *IEEE Robotics & Automation Magazine* (2010).

[15] Sina Darabi, Negin Mahani, Hazhir Baxishi, Ehsan Yousefzadeh-Asl-Miandoab, Mohammad Sadrosadati, and Hamid Sarbazi-Azad. 2022. NURA: A Framework for Supporting Non-Uniform Resource Accesses in GPUs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2022).

[16] Rina Dechter and Judea Pearl. 1985. Generalized Best-First Search Strategies and the Optimality of A. *Journal of the ACM (JACM)* (1985).

[17] Christer Ericson. 2004. *Real-Time Collision Detection.*

[18] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *International Symposium on Microarchitecture (MICRO)*.

[19] Matthew Evett, James Hendler, Ambuj Mahanti, and Dana Nau. 1995. PRA*: Massively Parallel Heuristic Search. *J. Parallel and Distrib. Comput.* (1995).

[20] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* (1968).

[21] Maurice Herlihy and J Eliot B Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture (ISCA)*.

[22] Kekib Irani and Yi-Fon Shih. 1986. Parallel A* and AO* Algorithms-An Optimality Criterion and Performance Evaluation. In *International Conference on Parallel Processing*.

[23] Hakbeom Jang, Yongjun Lee, Jongwon Kim, Youngsok Kim, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. 2016. Efficient Footprint Caching for Tagless Dram Caches. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[24] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A Scalable Architecture for Ordered Parallelism. In *International Symposium on Microarchitecture (MICRO)*.

[25] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. 1996. Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. *IEEE Transactions on Robotics and Automation* (1996).

[26] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. 2009. Scalable, Parallel Best-First Search for Optimal Sequential Planning. In *International Conference on Automated Planning and Scheduling*.

[27] Dan Klein and Christopher D Manning. 2003. A* Parsing: Fast Exact Viterbi Parse Selection. In *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*.

[28] Yoshiaki Kuwata, Justin Teo, Gaston Fiore, Sertac Karaman, Emilio Frazzoli, and Jonathan P How. 2009. Real-Time Motion Planning with Applications to Autonomous Urban Driving. *IEEE Transactions on control systems technology* (2009).

[29] Steven M LaValle et al. 1998. Rapidly-Exploring Random Trees: A New Tool for Path Planning. (1998).

[30] Jiaoyang Li, Zhe Chen, Daniel Harabor, P Stuckey, and Sven Koenig. 2021. Anytime Multi-Agent Path Finding Via Large Neighborhood Search. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

[31] Shiqi Lian, Yinhe Han, Xiaoming Chen, Ying Wang, and Hang Xiao. 2018. Dadu-P: A Scalable Accelerator for Robot Motion Planning in a Dynamic Environment. In *Design Automation Conference (DAC)*.

[32] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, et al. 2012. Scale-Out Processors. (2012).

[33] Samuel Moore. 2019. 3 New Chips to Help Robots Find Their Way Around. *IEEE Spectrum* (2019).

[34] Sean Murray, Will Floyd-Jones, George Konidaris, and Daniel J Sorin. 2019. A Programmable Architecture for Robot Motion Planning Acceleration. In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*.

[35] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. 2016. The Microarchitecture of a Real-Time Robot Motion Planning Accelerator. In *International Symposium on Microarchitecture (MICRO)*.

[36] Sean Murray, Will Floyd-Jones, Ying Qi, Daniel J Sorin, and George Dimitri Konidaris. 2016. Robot Motion Planning on a Chip. In *Robotics: Science and Systems.*

[37] Mike Phillips, Maxim Likhachev, and Sven Koenig. 2014. PA*SE: Parallel A* for Slow Expansions. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

[38] Ira Pohl. 1970. Heuristic Search Viewed As Path Finding in a Graph. *Artificial intelligence* (1970).

[39] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *International Symposium in Computer Architecture (ISCA)*.

[40] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently Prefetching Complex Address Patterns. In *International Symposium on Microarchitecture (MICRO)*.

[41] Egor Shipovalov and Valentin Pryanichnikov. 2020. Scalable State Space Search on the GPU with Multi-Level Parallelism. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*.

[42] Nathan R Sturtevant. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)* (2012).

[43] Richard Anthony Valenzano, Nathan Sturtevant, Jonathan Schaeffer, Karen Buro, and Akihiro Kishimoto. 2010. Simultaneously Searching with Multiple Settings: An Alternative to Parameter Tuning for Suboptimal Single-Agent Search Algorithms. In *International Conference on Automated Planning and Scheduling*.

[44] Vincent Vidal, Lucas Bordeaux, and Youssef Hamadi. 2010. Adaptive K-Parallel Best-First Search: A Simple but Efficient Algorithm for Multi-Core Domain-Independent Planning. In *Annual Symposium on Combinatorial Search*.

[45] Zishen Wan, Bo Yu, Thomas Yuang Li, Jie Tang, Yuhao Zhu, Yu Wang, Arijit Raychowdhury, and Shaoshan Liu. 2020. A Survey of FPGA-Based Robotic Computing. *arXiv preprint arXiv:2009.06034* (2020).

[46] Christopher Makoto Wilt and Wheeler Ruml. 2012. When Does Weighted A* Fail?. In *SOCS*.

[47] Kai M Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. 2010. OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems. In *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*.

[48] Victor A Ying, Mark C Jeffrey, and Daniel Sanchez. 2020. T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware. In *International Symposium on Computer Architecture (ISCA)*.

[49] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. 2020. Building the Computing System for Autonomous Micromobility Vehicles: Design Constraints and Architectural Optimizations. In *International Symposium on Microarchitecture (MICRO)*.

[50] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-Based Computations Through Principled Speculation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[51] Yichao Zhou, Wei Xu, Bruce R Donald, and Jianyang Zeng. 2014. An Efficient Parallel Algorithm for Accelerating Computational Protein Design. *Bioinformatics* (2014).