Bridging Real-Time Robotics and Computer Architecture

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Mohammad Bakhshalipour

B.S., Computer Engineering, Sharif University of Technology
M.S., Computer Engineering, Sharif University of Technology

Carnegie Mellon University
Pittsburgh, PA

May 2024

Acknowledgements

I want to express my heartfelt thanks to my advisor, Professor Phillip Gibbons. Throughout my PhD journey, I have always felt incredibly lucky to have Phil as my advisor. He's gone above and beyond in supporting me, for which I'm deeply grateful. He not only set me on an engaging and challenging research path but also showed immense kindness.

I also want to express my thanks to Professors Saugata Ghose and Nathan Beckman, with whom I had the opportunity to work. Saugata's support during my transition to Phil's team was incredibly valuable, and his guidance up to his last day at CMU was deeply appreciated. Though my time with Nathan was brief, his insights on research approaches have profoundly influenced my work.

A big thank you to Professors Akshitha Sriraman, Zachary Manchester, and Vijay Janapa Reddi for graciously agreeing to be part of my defense committee. Their valuable feedback and insights were instrumental in refining my work and guiding me towards successful completion.

Collaborating with Professors Maxim Likhachev and Akshitha Sriraman has been a highlight of my time at CMU, and I'm grateful for their insights. Thanks to my colleagues and lab-mates—Dominic Chen, Elliot Lockerman, Mohamad Qadri, Dominic Guri, Yiwei Zhao, Valerie Choung, and others—for enriching collaborations. A special shoutout to the PDL community for providing opportunities to present my work to a broader audience. A special thanks to Jason Boles, our incredible sysadmin, without whom I wouldn't meet any of the deadlines.

My entry into computer architecture research was guided by my Masters' advisors, Professors Hamid Sarbazi-Azad and Pejman Lotfi-Kamran. Pejman played a pivotal role in my development, welcoming me into this field despite my limited background, and I owe much of what I know about computer architecture to him. Hamid was more than an advisor; his support and guidance extended well beyond the academic realm.

I extend my gratitude to the Apple PhD Fellowship, the Carnegie Institute of Technology Dean's Fellowship, and the James Sprague Presidential Fellowship for their financial support throughout my PhD.

My adaptation to a new country over the past years was greatly eased by the support of my dear friends. I'm profoundly grateful to Hodjat Asghari, Ata Fattahi, Sara Mahdizadeh, and Armin Vakil for their unwavering friendship in every circumstance, to Sajjad Dadashi for his warm hospitality upon my arrival in Pittsburgh, to my roommate, Julian Tuminaro, for putting up with me, and to Vahid Hassanzade and Mohammad Reza Takallouie for their care upon my hospitalization. Thanks to my friends at EGO, Maia Blanco, Nikhil Agarwal, and others, for all the memorable events. And to everyone in Pittsburgh who joined in on the parties, travels, and games: a huge thank you!

And lastly, but most importantly, to the love of my life, Fatemeh Golshan. Fatemeh, your compassion and enduring love have been the heart of my life. Your selflessness and patience through the endless days and nights of work have been the foundation I've leaned on. I am immeasurably grateful for everything you are and all that you've done.

Abstract

Robotics technology, poised to play a pivotal role in shaping future societal frameworks, is projected to reach a deployment of 20 million units and a market valuation of US$70 billion by the end of this decade. Despite its growing significance, there remains a pronounced gap between the fields of robotics and computer architecture research. This thesis endeavors to bridge this gap, contributing to the advancement of real-time robotic systems through two primary initiatives: *(i)* the development of comprehensive, open-source benchmark suites coupled with systematic performance studies, and *(ii)* the devising of efficient computer architectures for robotics.

Firstly, the thesis introduces extensive benchmark suites for robotics tasks and applications, meticulously developed from scratch to address the critical need for performance-focused, comprehensive benchmarking environments in systems and hardware research. These suites are accompanied by thorough performance analyses, delving into the computational behaviors and architectural implications of the applications. Employing advanced optimization techniques such as compile-time calculations, data prefetching, and manual code vectorization, the benchmarks offer insights into execution efficiency and the effective utilization of hardware resources. Moreover, the benchmarks are designed to be simulator-friendly, ensuring compatibility with architectural simulators, which is crucial for the predominantly simulator-based methodologies used in hardware research. Such suites and performance studies are instrumental in deepening the understanding and improving the synergy between robotics and computer architecture, facilitating the development of more efficient and effective robotic systems. Through this approach, the thesis contributes significantly to the field by enhancing the benchmarking tools available for evaluating and optimizing robotic applications within computer architectures.

Secondly, the thesis proposes efficient computer architectures tailored for robotics, focusing on the development of application-specific hardware accelerators and domain-specific processors. These hardware accelerators are specifically designed to optimize crucial robotic kernels, such as collision detection, enhancing their efficiency significantly. Conversely, the domain-specific processors, while not exclusively tailored for individual kernels, are optimized to deliver enhanced performance across a broader spectrum of robotic applications. The process begins with extensive profiling of robotic applications to identify performance bottlenecks. This is followed by the design of targeted architectural solutions aimed at accelerating these key operations, thereby improving the overall execution times of the systems. The empirical results demonstrate significant speedup potentials, achieving up to 41.4 times with application-specific hardware accelerators and 3.8 times with domain-specific processors.

Collectively, these contributions not only propel the performance capabilities of real-time robotic systems but also establish a groundwork for ongoing and future research at the critical nexus of robotics and computer architecture.

Contents

Contents **v**

List of Tables **vii**

List of Figures **vii**

1 Introduction **1**

1.1 Thesis Statement 1

1.2 Challenges 2

1.3 Contributions 3

1.4 Outline 6

2 Background **8**

2.1 Robotics Software Pipeline 8

2.2 Terminology 9

2.3 The "Real-Time" Performance 10

2.4 The Role of Computation in Power and Energy Consumption 11

3 Motivation **12**

3.1 The Importance of Robotics 12

3.2 The Importance of Computation Performance for Robotics 14

3.3 The Thesis's Methodology 17

4 RTRBench: Benchmarking Robotics Tasks **20**

4.1 Analysis Framework 21

4.2 RTRBench Tasks 21

4.3 Implementation Details 37

4.4 Comparison with Open-Source Repositories 38

4.5 Chapter Summary . . . 39

5 RoWild: Architectural Implications of Robotics on Modern Hardware 40

5.1 Existing Work . . . 41
5.2 The RoWild Approach . . . 42
5.3 The Modeled End-to-End Applications . . . 47
5.4 Architectural Implications of Robotics . . . 60
5.5 Chapter Summary . . . 65

6 RACOD: Algorithm/Hardware Co-design for Mobile Robot Path Planning 66

6.1 Motivation . . . 68
6.2 Runahead Collision Detection . . . 72
6.3 Methodology . . . 77
6.4 Evaluation . . . 78
6.5 Runahead Multithreading . . . 86
6.6 RA*: Speculative Parallelism for A* with Slow Expansions . . . 88
6.7 RACOD and RA* and in The Context of Related Work . . . 104
6.8 Chapter Summary . . . 106

7 Tartan: Microarchitecting a Robotic Processor 107

7.1 The Need for Efficient Robotics CPU . . . 109
7.2 The Need for Efficient Memory Systems . . . 110
7.3 Performance Study . . . 110
7.4 Oriented Vector Loads . . . 113
7.5 Approximate-Acceleration . . . 115
7.6 Efficient NNS in High-Dimensional Spaces . . . 120
7.7 Intra-Application Cache Partitioning . . . 123
7.8 Evaluation . . . 126
7.9 Chapter Summary . . . 133

8 Conclusion and Future Directions 135

8.1 Extending The Benchmark Suite . . . 136
8.2 Quantitative Metrics for Evaluating Platforms . . . 136
8.3 Beyond Performance . . . 137

Bibliography **139**

List of Tables

1.1 Feature comparison of benchmarks developed in this thesis with those from prior works. 3
1.2 The workloads included in the benchmark suites. 4

4.1 RTRBench's tasks and their key characteristics. 22

5.1 Feature comparison of related work and RoWild. 42
5.2 RoWild's workloads. 43
5.3 The evaluated compute platforms. 45
5.4 The modeled end-to-end applications. 47

6.1 Oriented bounded box configuration encoding. 73
6.2 Design parameters of the collision detection accelerator. 78

7.1 Application parameters. 111
7.2 The neural network workloads evaluated. 128
7.3 Different neural processing unit configurations evaluated. 129
7.4 Overhead breakdown. 133

List of Figures

1.1 Bottleneck analysis and speedup achieved by efficient computer architectures. 5

2.1 Robots' computation pipeline. 8

3.1 Global market value of robotics estimated by different sources. 13
3.2 The evolution of low-end and high-end robotics processors over time. 16
3.3 Mapping compute-intensive robotics operations to different architectural concepts. 18

4.1 Monte Carlo localization. 23
4.2 SLAM using Extended Kalman Filter. 24
4.3 3D reconstruction in dynamic scenes. 25
4.4 2D path planning. 26
4.5 3D path planning. 27
4.6 Catching a moving target. 28
4.7 Robotic arm motion planning. 28
4.8 Synthetic maps for evaluating the robotic arm. 29
4.9 Arm manipulator planning by the RRT algorithm. 30
4.10 A rewiring example with RRT*. 31
4.11 Post-processing the path found by RRT. 31
4.12 Blocks world problem. 32
4.13 Firefighter robots. 33
4.14 Dynamic movement primitives. 34
4.15 Model predictive control. 35
4.16 Ball-throwing robot. 35
4.17 Rewards over time using CEM. 36
4.18 Rewards over time using BO. 36
4.19 An example of a help message. 37
4.20 Performance comparison of different libraries. 38

5.1 An example of a help message. 45
5.2 DeliBot operating in CMU's Wean Hall. 49
5.3 PatrolBot operating on CMU campus. 51
5.4 MoveBot operating in a synthetic environment. 52
5.5 HomeBot operating in a benchmark environment. 55
5.6 FlyBot operating in a benchmark environment. 57
5.7 CarriBot operating in a benchmark environment. 58
5.8 Vectorization's impact on low-end and high-end CPUs. 61
5.9 Performance scaling hits the memory wall. 62

5.10 Prefetching's impact on low-end and high-end CPUs. 63
5.11 Data movements breakdown. 64

6.1 2D mobile robot path planning, oriented bounded box, and path graph search. 68
6.2 Hardware realization of collision detection accelerator. 74
6.3 2D navigation in the wild. 79
6.4 Cone-like patterns in a Boston snapshot, with a runahead of 32. 80
6.5 A map of the environment and the performance improvement. 80
6.6 The modeled application and the performance improvement of hardware acceleration. 81
6.7 Speedup sensitivity to CPU-accelerator communication latency. 82
6.8 Prediction accuracy/coverage with different runaheads. 82
6.9 Division-of-labor, varying the number of accelerators. 84
6.10 RACOD's effectiveness with WA* and different heuristics. 85
6.11 L_0 cache hit ratio with varying its size. 85
6.12 Prediction throttling impact. 86
6.13 Performance comparison of different platforms and configurations. 87
6.14 Graph search using A*. 89
6.15 2D path planning example. 91
6.16 Sequitur's grammar with an example. 93
6.17 Speedup comparison of methods. 95
6.18 Performance comparison of different predictors. 97
6.19 Harnessing SLP on random maps. 99
6.20 SLP in real-world and random environments. 99
6.21 RA*'s speedup with coarsening the resolution. 100
6.22 Speedup comparison of methods for WA* 100
6.23 Workload distribution and thread utilization with different numbers of threads. 101
6.24 Projected speedup. 103

7.1 Execution time breakdown and bottleneck analysis. 112
7.2 Tartan's oriented vectorization. 113
7.3 Tartan's neural processing unit. 117
7.4 Nearest-neighbor search with locality-sensitive hashing. 121
7.5 Tartan's fuzzy cache partitioning with an example application. 123
7.6 Cache sets miss density with motion planning progress. 124

7.7 Oriented access patterns and different vectorization methods. 127
7.8 Neural acceleration of robotics. 129
7.9 Nearest-neighbor search with different approaches. 130
7.10 Different prefetching approaches. 130
7.11 Fuzzy cache partitioning with different parameters. 131
7.12 Tartan's performance with legacy and optimized software. 132

Chapter 1

Introduction

The rapid advancement in robotics technology is profoundly transforming our world. With an estimated 20 million robots projected by 2030 [106] and a market valuation poised to reach US$70 billion by 2025 [83], the significance of robotics in future societal frameworks is undeniable. For widespread adoption, robots must be designed to operate autonomously and execute complex artificial intelligence (AI) tasks in real-time within diverse real-world settings [181].

The role of computer hardware and architecture is critical in enabling real-time functionality in robotics. This is evidenced by the integration of robot-specific hardware accelerators in the architecture of modern edge processors. Recent robotics platforms [136, 137, 165, 171] incorporate hardware accelerators optimized for operations such as tree-extension and ray-casting, which are extensively used in robotics. Intel's multi-robot system [136] encompasses a full-fledged "Robot SoC."

Surprisingly, the computer architecture research community has under-explored the field of robotics. Despite the escalating prominence of robots in our technologically driven society and the burgeoning emergence of industrial "robotic processors," a large gap persists between robotics and the computer architecture research community. This is evident from the limited number of related publications in architecture conferences. For example, in 2022 (2023), among the more than 300 papers presented at the leading computer architecture conferences such as ISCA, MICRO, HPCA, and ASPLOS, only three (seven, respectively) were dedicated to robotics.

1.1 Thesis Statement

This thesis contends that the gap between robotics and computer architecture research has engendered an innovation barrier. This barrier impedes the exploration, development, and evaluation of architecture-level techniques that could significantly advance robotics, potentially yielding improvements such as speedups of an order of magnitude or more. As a consequence of this barrier, there exists a pronounced *mismatch* between the requirements of contemporary robotics workloads and the design of mainstream processors.

Thesis Statement

A prevailing gap exists between the fields of robotics and computer architecture research, resulting in a mismatch between the architectural design of mainstream processors and the demands of robotics workloads. This thesis (i) endeavors to bridge this robotics-architecture gap through the introduction of frameworks that facilitate the study of robotics within the realm of computer architecture, and (ii) propounds architectural solutions to rectify the workload-architecture mismatch, ultimately leading to substantial enhancements in robotics performance.

1.2 Challenges

This thesis endeavors to bridge the gap between the robotics and computer architecture research, focusing on enhancing the development of real-time robotic systems by addressing the following key challenges.

1.2.1 The Lack of a Comprehensive Benchmark Suite

A primary factor contributing to the robotics-computer architecture research divide is the absence of a comprehensive, open-source benchmark suite. Current benchmark suites are plagued by significant shortcomings, rendering them unsuitable for architectural studies. Due to this deficiency, the computational behaviors of robotics tasks remain obscure to the computer architecture community. Consequently, the limited research intersecting robotics and computer architecture often encompasses only a single [186, 200] or a few applications [139, 146, 220] in their analyses, leaving the impact on other applications unexplored.

1.2.2 The Lack of a Systematic Performance Study

Another factor exacerbating the gap between robotics and architecture is the absence of systematic performance studies. This gap results in the performance requirements and architectural implications of robotics workloads remaining largely uncharted to the architecture community.

1.2.3 The Mismatch between Robotics Workloads and Modern Processors

A direct consequence of the robotics-architecture divide is the significant mismatch between the design of modern processors and the specific demands of robotics workloads. Robotics computations exhibit unique characteristics that differ markedly from those of workloads (e.g., cloud, desktop) for which most contemporary processors are optimized. This mismatch underscores the need for targeted architectural innovations to better cater to the requirements of robotics applications.

1.3 Contributions

This thesis seeks to bridge the divide between robotics and computer architecture research through two primary objectives, detailed as follows.

1.3.1 Benchmarking Robotics

The initial goal of this thesis is the development of comprehensive, open-source benchmark suites. This entails creating benchmarks for individual robotics tasks (*RTRBench*; see Chapter 4) as well as for complete end-to-end robotics applications (*RoWild*; see Chapter 5). Post-development, this thesis conducts an in-depth analysis of the computational behaviors, performance demands, and architectural implications of the developed robotics workloads.

The development of benchmark suites involves essential considerations that make them suitable for systems and hardware research. These benchmarks are designed to be comprehensive, incorporating a diverse range of robotic workloads to ensure their applicability across various robotic applications. A critical focus on performance underpins our approach: we develop the codes *from scratch* using native programming languages. This is complemented by the use of industry-standard profiling tools and advanced software techniques to optimize execution speed and efficiency. Techniques such as compile-time calculations, data prefetching, and manual code vectorization are employed to significantly enhance performance, distinguishing our benchmarks from other available robotic repositories. Moreover, these suites are designed to be simulator-friendly, facilitating seamless integration with architectural simulators—a key feature for hardware research that relies heavily on simulation tools.

In contrast to our comprehensive approach, previous benchmarking efforts often fall short in providing crucial features necessary for studying robotics within computer architecture. Our contributions, detailed in the upcoming chapters and highlighted in Table 1.1, include *RTRBench* and *RoWild*. These benchmarks offer essential features for robotics-architecture studies that set them apart from earlier works.

Table 1.1: Feature comparison of benchmarks developed in this thesis with those from prior works. More ✔ is better.

Paper/Repository	Scope	End-to-End	High-Perf.	Multi-Platform	Open-Source	Simulator-Friendly	Versatile & Modular	System-Level Analysis
Lin et al. [184] Yu et al. [253]	Self-Driving Cars	✔	✔✔	✔✔				✔
MAVBench [124]	Drones	✔	✔	✔	✔			✔
One-off [88, 162]	Single Task				✔		✔	
ROS [80]	Broad				✔		✔	
Educational [14, 66]	Broad				✔			
RTRBench (§4)	Broad		✔		✔	✔	✔	
RoWild (§5)	Broad	✔	✔✔	✔✔	✔	✔	✔	✔✔

Importantly, the selection of tasks and algorithms for the benchmark suites is based on an analysis of dozens of real-world robots, including a diverse range from arm manipulators and home-cleaning robots to self-driving vehicles. This comprehensive analysis ensures that the task selection is both broad and relevant. Additionally, the suites incorporates state-of-the-art research algorithms, such as deep learning-based pathfinding, positioning it as a forward-looking benchmark suite for robotics.

Table 1.2 showcases the tasks and the algorithms implemented to execute them, with symbols indicating the real-world robots that either come pre-packaged with the software where the algorithm is used or are custom-programmed in the literature to perform specific missions.[1] The following chapters, specifically §4.2 and §5.2, provide detailed explanations of the tasks and algorithms, including insights into selection criteria and implementation considerations.

Task	Algorithms Implemented
State Estimation	MCL$^{\dagger\ddagger\Upsilon\P\eta\sigma}$ [254], AMCL$^{\zeta}$ [239]
Localization and Mapping	EKF$^{\Upsilon\S\tau\phi\Lambda}$ [240], Fast$^{\ddagger\P\zeta}$ [195], Graph$^{\vartheta\varphi}$ [225], ORB$^{\varsigma\rho}$ [129]
Scene Reconstruction	Point-Based Fusion$^{\dagger\ddagger\Upsilon\kappa\upsilon}$ [249]
Grid Generation	Probabilistic Occupancy Map$^{\dagger\kappa\vartheta}$ [151]
Object Detection	MobileNet-SSD$^{\ddagger\Upsilon\rho\chi\Lambda}$ [252]
2D/3D Navigation	WA$^{\star\dagger\P\zeta\eta\alpha\Pi\Lambda}$ [216], GA* [256], RA* [118], IDA$^{\star\varsigma}$ [174] , DQN [194]
Timed Search	WA$^{\star\dagger\P\zeta\eta\alpha\Pi\Lambda}$ [216] + Backward Dijkstra [120]
2D/3D n-DoF Search	RRT$^{\dagger\ddagger\Upsilon\kappa}$ [148], RRT$^{\star\Upsilon\omega\gamma\delta}$ [152], PRM$^{\Upsilon\vartheta\psi}$ [234], Shortcut [153]
Reactive Planning	Behavior Tree$^{\dagger\lambda\vartheta\phi}$ [154]
Task Scheduling	Symbolic Planning$^{\S\vartheta\delta}$ [122]
Motion Control	MPC$^{\Upsilon\Pi\vartheta\dagger\kappa\alpha\psi\varepsilon}$ [144] PID$^{\Upsilon\vartheta\zeta\theta\phi}$ [247], PP$^{\ddagger\Upsilon\kappa\phi}$ [209]
Parameter Learning	DMP$^{\Upsilon\vartheta\xi\alpha\theta\iota}$ [175], CEM$^{\S\lambda}$ [212], BO$^{\rho}$ [163]

Table 1.2: The workloads included in the benchmark suites.

Real-World Robots:

†Spot [9]
‡DJI Phantom [61]
$^{\Upsilon}$LoCoBot [103]
§Atlas [8]
¶AMR [72]
$^{\Pi}$AscTec Pelican [6]
$^{\zeta}$Roomba 980 [78]
$^{\upsilon}$Roomba i7+ [79]
$^{\eta}$Husky [26]
$^{\vartheta}$YuMi [101]
$^{\kappa}$Jackal [44]
$^{\xi}$LBR [52]
$^{\rho}$Pepper [85]
$^{\sigma}$TurtleBot [94]
$^{\varsigma}$TurtleBot3 [95]
$^{\tau}$MiR [54]
$^{\phi}$AG [50]
$^{\Lambda}$Pioneer 3-DX [64]
$^{\varphi}$UR10e [98]
$^{\chi}$TALOS [87]
$^{\psi}$KR60-3 [51]
$^{\omega}$Grizzly [12]
$^{\gamma}$Skydio [84]
$^{\delta}$M-200iA/2300 [20]
$^{\varepsilon}$PerceptIn [252]
$^{\alpha}$Boxbot [10]
$^{\theta}$UR5e [99]
$^{\iota}$Franka Panda [22]
$^{\lambda}$PAL TIAGo [92]

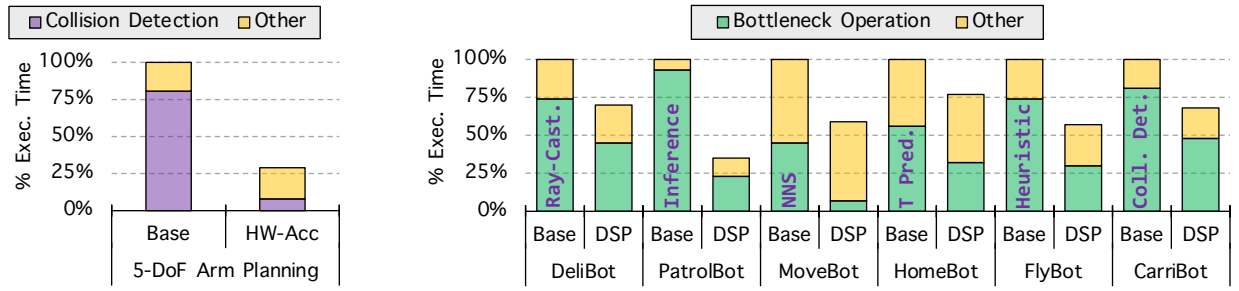
1.3.2 Devising Efficient Computer Architectures for Robotics

The second aim of this thesis is the design and implementation of efficient computer architectures tailored for robotics, thereby augmenting their performance and facilitating the realization of real-time robotics on a broader scale. This objective includes the creation of *(i)* application-specific hardware accelerators (Chapter 6) and *(ii)* domain-specific processors (Chapter 7). The former is dedicated to executing crucial robotic kernels (such as collision detection) with optimal efficiency via the custom design of hardware circuits specifically for these tasks. The latter revolves around the development of processors that, while not exclusively tailored for particular kernels, deliver enhanced performance

[1] In some cases, the identification of these algorithms is based on the analysis of other works. However, it should not be assumed that the robots listed are certainly or exclusively using these algorithms.

for a more extensive range of robotic applications. Hardware accelerators are aimed at achieving peak performance for specialized tasks, whereas domain-specific processors strike a balance between performance and versatility, thus accommodating a wider spectrum of applications.

Figure 1.1 showcases the results of this aim of the thesis. In either approach, we first perform extensive profiling of the robotic applications running in different settings to identify bottleneck operations. After identifying the bottlenecks, we devise efficient architectural solutions for accelerating the bottlenecks and thereby the end-to-end execution time.



(i) Application-Specific Hardware Accelerator *(ii)* Domain-Specific Processor

Figure 1.1: Bottleneck analysis and speedup achieved by efficient computer architectures.

Figure 1.1 (left) showcases the results with the first approach, i.e., application-specific hardware accelerator. The results show the end-to-end execution time of a case-study robot with a state-of-the-art Baseline robotic processor and the HardW-Accelerator we devise in this thesis for costly collision detection operations. The results also show the contribution of the collision detection operation and how significantly it is reduced with the proposed approach. The details of the approach and the evaluation methodology are provided in Chapter 6.

Figure 1.1 (right) showcases the results with the second approach, i.e., domain-specific processor. The results show the end-to-end execution time of different robots with a state-of-the-art Baseline robotic processor and the Domain-Specific Processor we devise in this thesis. The results also show the contribution of bottleneck operations identified in different applications. The details of the approach and the evaluation methodology are provided in Chapter 7.

As the results indicate, the application-specific hardware accelerator is able to offer a significant level of speedup for the target operation. In the showcased results, the devised hardware accelerator achieves a 10× speedup in the target operation, i.e., collision detection, which translates to a considerable end-to-end speedup. On the other hand, the devised domain-specific processor's speedup on target operations (i.e., different identified bottlenecks in different applications) averages around 2.6×; while it is significant, it is smaller than the speedup of the application-specific hardware accelerator; on the other hand, it benefits a wide spectrum of applications, striking a balance between performance and generality.

1.4 Outline

The rest of this thesis is organized as follows.

- Chapter 2 lays the foundation for subsequent discussions in this thesis by providing essential background on robotics tasks, algorithms, and applications. It offers a comprehensive overview of pertinent topics in the field of robotics, setting the stage for deeper exploration.
- Chapter 3 introduces the motivation behind the thesis's goal of bridging the fields of robotics and computer architecture. This chapter presents data highlighting the significance of robotics computation, explores the evolution of robotics software and hardware, and envisions their future requirements. Additionally, it discusses the thesis's approach and the underlying philosophy guiding the research.
- Chapter 4 describes *RTRBench*, a benchmark suite developed for *individual* robotics tasks. It discusses their computational characteristics and architectural implications. *RTRBench* contributes to the first initiative of the thesis, namely, benchmarking robotics.
- Chapter 5 introduces *RoWild*, a benchmark suite coupled with a systematic performance study focusing on *end-to-end* robotics applications. *RoWild* also contributes to the first initiative of the thesis: benchmarking robotics. *RoWild* won the best paper award at ACM SIGMETRICS 2024.
- In Chapter 6, the thesis presents *RACOD*, an algorithm/hardware co-design for mobile robot path planning. *RACOD* introduces an *application-specific* hardware accelerator designed to speed up frequent collision detection operations in robotics. *RACOD* contributes to the second initiative of the thesis, which is devising efficient computer architectures for robotics.
- Chapter 7 presents *Tartan*, a *domain-specific* processor designed for broader robotics applications. Unlike *RACOD*, which focuses on a single task, *Tartan* targets a wider range of tasks and computations utilized in robotics. *Tartan* contributes to the second initiative of the thesis: devising efficient computer architectures for robotics.
- Finally, Chapter 8 summarizes the key findings and contributions of the thesis and suggests directions for future research.

This thesis includes content from the research papers published during the course of this study. The table below lists these publications and the implementations that have been made available as open-source.

Chapters and The Materials They Utilize

Chapter	**Publications**	**Source Code**
Chapter 4	[116]	https://github.com/cmu-roboarch/rtrbench
Chapter 5	[113, 114]	https://github.com/cmu-roboarch/rowild
Chapter 6	[112, 118]	https://github.com/cmu-roboarch/runahead-astar
Chapter 7	[115]	https://github.com/cmu-roboarch/tartan

Chapter 2

Background

This chapter establishes the groundwork necessary for understanding the subsequent discussions in this thesis by providing an in-depth overview of pertinent aspects in the field of robotics.

2.1 Robotics Software Pipeline

The software pipeline of a generic autonomous robot, as illustrated in Figure 2.1, consists of three integral stages: *perception*, *planning*, and *control*.

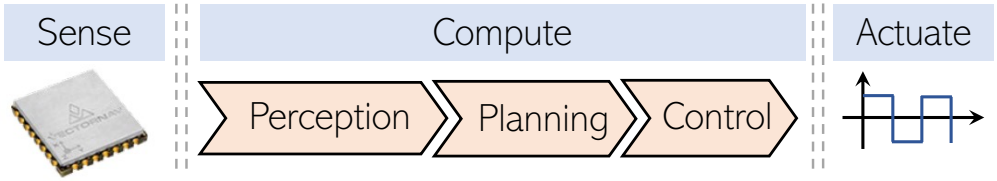


Figure 2.1: Robots' computation pipeline.

2.1.1 Perception

The perception stage is tasked with comprehending the state of the environment and the robot itself. It processes *raw* data from sensors to *infer* the robot's state (such as location and orientation) and its surroundings (like identifying obstacles). The process of determining the robot's state is termed *localization*, while understanding the environment is referred to as *mapping*.

2.1.2 Planning

The planning stage is responsible for charting a course from the robot's current location to a designated target. Utilizing the output from the perception stage, the planner discerns the positions of obstacles and navigates the environment

to devise an efficient and *collision-free* path.

2.1.3 Control

The control stage generates commands to adhere to the path established by the planning stage. It calculates the necessary dynamics (like velocity and acceleration) for the robot to *efficiently* follow the path and subsequently issues the appropriate commands to the robot's actuators.

2.1.4 The Implementation of Every Stage

The algorithms utilized in each stage are contingent on the specific application and environment. For example, robots equipped with cameras employ computer vision algorithms, whereas those with LIDAR use point cloud processing. The choice of algorithm also depends on operational priorities, such as battery-powered drones optimizing for the shortest path to maximize range, and search-and-rescue robots prioritizing the safest path.

2.1.5 Identifying the Performance Bottleneck

The performance bottleneck in a robotic system varies depending on the robot, mission, and environment. For instance, in a home assistant robot searching for an item in a cluttered space, perception might be the bottleneck. In contrast, for a stationary robotic arm with high degrees of freedom, planning might pose the greatest challenge. Similarly, for an autonomous vehicle navigating curvy roads, control could be the critical bottleneck.

Previous studies have identified different bottlenecks: Yu et al. [253] found perception to be the limiting factor in their application, while Murray et al. [200] identified planning, and Sacks et al. [220] pinpointed control. Boroujerdian et al. [125] noted that the performance bottleneck can shift depending on the mission of an unmanned aerial vehicle (UAV), such as package delivery versus search and rescue operations.

2.2 Terminology

Below, we define some of the terminology we employ throughout this thesis.

- **Robotic Tasks:** The term "robotic tasks" pertains to *generic* high-level operations such as scene understanding and pathfinding. The range of these operations is finite within the field of robotics, but the ways they are carried out can vary from one robot to another. For instance, both autonomous vehicles and home-assistant robots undertake the task of pathfinding. However, the employed algorithms and the constraints could significantly differ in these two scenarios.

- **Robotic Algorithms:** By "robotic algorithms," we refer to specific algorithms implemented to carry out robotic tasks; e.g., the A* algorithm [216] for pathfinding. There are hundreds, or even thousands, of such algorithms proposed in the robotics community to accomplish robotic tasks.
- **Robotic Applications:** The term "robotic applications" denotes the end-to-end software pipeline of a specific robotic system, for instance, an autonomous drone. The scope of robotic applications is vast, extending from self-driving vehicles to home-assistant robots, and the spectrum continues to widen with future advancements in robotics.

2.3 The "Real-Time" Performance

The determination of what constitutes "real-time" performance for a robot is contingent upon the specific application it serves. In the context of robotic systems, real-time performance is characterized by adhering to predefined maximum latency (i.e., deadline) constraints at each stage of operation. These latency constraints are instrumental in ensuring the robot's responsiveness and effectiveness in its designated tasks. Notably, the acceptable latency values can vary considerably depending on the nature of the application.

For instance, within the realm of perception tasks, prior research has identified latency constraints such as 30ms [145] and 200ms [133] as indicative of real-time operation. Conversely, in the realm of planning tasks, earlier studies have reported stringent requirements, with deadlines measured in sub-milliseconds [200], while others have found 250ms [126] to be an acceptable threshold.

This diversity in latency constraints underscores the necessity of architecting robotic systems to optimize performance across all stages. Ensuring that each component of the system can meet the real-time requirements, irrespective of the application's demands, is a critical engineering challenge.

One of the fundamental trade-offs encountered in the development of robotic systems is the delicate balance between accuracy and execution time. Virtually all robotic tasks grapple with this trade-off: enhancing accuracy typically necessitates increased computational resources and, consequently, results in longer execution times. This trade-off is a key consideration in the design and operation of both commercial robots and research-oriented robotic proposals.

For instance, the EureCar Turbo, a self-driving car based on the Hyundai Veloster Turbo platform, makes a deliberate trade-off by employing an intentionally *inaccurate* collision detection mechanism. This compromise is made to achieve acceptable execution times [180]. The implication here is that optimizing the underlying hardware and software platforms for different robotic kernels is imperative. Such optimization enables the attainment of real-time performance without compromising significant accuracy.

2.4 The Role of Computation in Power and Energy Consumption

Computation is *not* a substantial power consumer in the context of robotic systems, as established by previous research [124, 125, 177, 191]. In the realm of robotics, the majority of power, exceeding 95%, is devoted to driving the mechanical components, thereby rendering the role of computation in overall power (not energy) consumption relatively inconsequential. For instance, in the case of the 3DR Solo Drone [1], the rotors account for a power demand exceeding 286 watts, whereas computational processes draw less than 13 watts [124].

When computation's power consumption becomes nearly negligible (i.e., $Power_{Total} \approx Power_{Rotors}$), the overall energy efficiency of a robotic system is closely linked to its computational *performance*. I.e., $Energy = Power \times t$, where *Power* pertains to the overall power consumption and t represents the duration of computational tasks—optimizing computational performance plays a pivotal role in enhancing overall energy efficiency.

Chapter 3

Motivation

This chapter elucidates the motivation behind the thesis's objective to bridge the fields of robotics and computer architecture. It provides data that underscore the critical importance of computation in robotics, examines the development of robotics software and hardware, and projects their future demands. Furthermore, this chapter delineates the thesis's methodology and the fundamental philosophy that underpins the research.

3.1 The Importance of Robotics

Robots are increasingly becoming integral to various sectors of our society, influencing aspects ranging from the economy [77] and healthcare [90] to agriculture [91] and the military [89]. Figure 3.1 illustrates the global market value of robotics, as estimated by several research firms such as Mordor Intel [73], Allied Market Research [75], Fortune Business Insights [28], INN [74], Grand View Research [27], SkyQuestt [25], and Statista [83], with values reported in billions of USD. The numbers in the graph report compound annual growth rate (CAGR).

Despite methodological differences among these firms and varied perspectives on what constitutes robotics, all indicate a robust market poised for substantial growth–from tens of billions currently to potentially hundreds of billions in the next decade–with CAGRs ranging from 10% to 27%. This projected growth is driven by a combination of technological advancements, economic factors, and broader adoption across diverse industries.

Technological innovations in artificial intelligence, machine learning, and sensor technology have drastically enhanced the capabilities and reduced the costs of robotic systems. These improvements have made robots more accessible and efficient, with industries such as manufacturing, healthcare, and logistics incorporating them to streamline operations, increase precision, and reduce human error. The rise of e-commerce, especially accelerated by the COVID-19 pandemic, has necessitated advanced solutions in warehousing and distribution, further boosting the robotics market. Additionally, demographic changes, such as aging populations in developed countries, have catalyzed the demand for service robots in personal care and medical assistance. Moreover, an increasing focus on sustainable practices has led

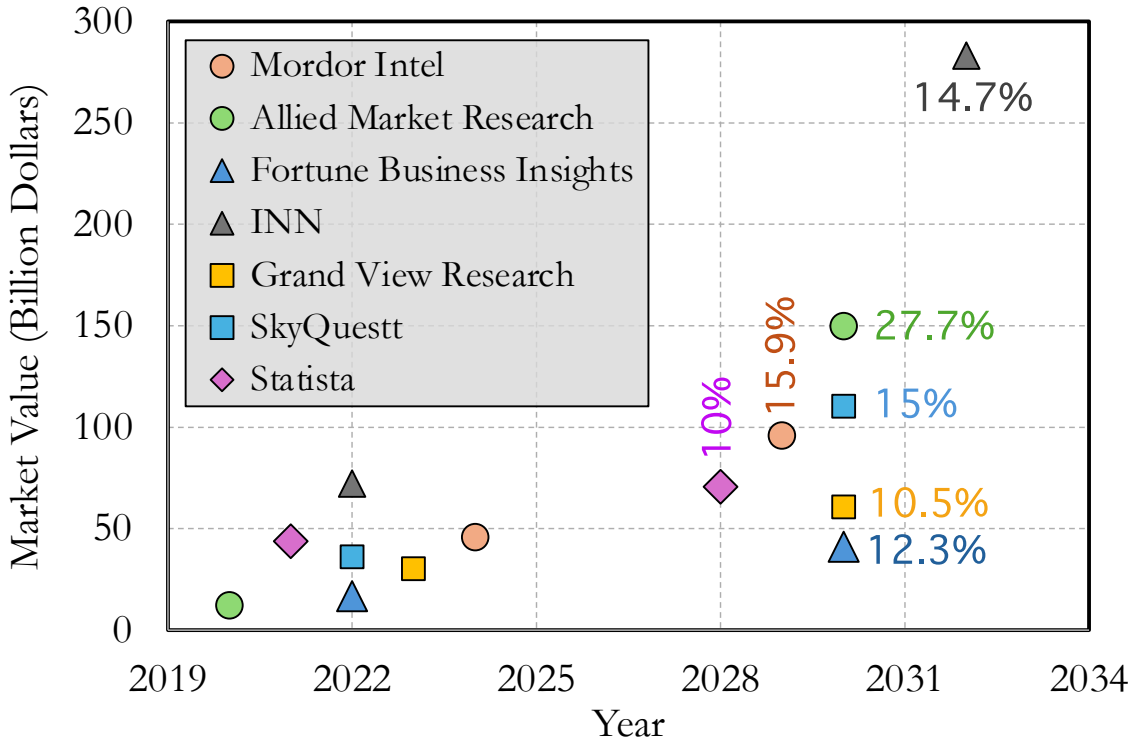


Figure 3.1: Global market value of robotics estimated by different sources.

businesses to employ green automation technologies that use robotics to improve efficiency while reducing environmental impacts. These factors collectively affirm a strong growth trajectory for the robotics market, solidifying its role as a pivotal component of the future global economy.

Consequently, governments worldwide are recognizing the strategic importance of robotics in fostering technological innovation and economic growth, leading to an array of support mechanisms including financial investments, regulatory frameworks, and research initiatives.

For instance, the US has announced initiatives to promote responsible AI and robotics innovation. This includes the National Science Foundation's allocation of $140 million to establish seven new National AI Research Institutes, broadening the network of organizations engaged in AI and robotics across nearly every state [18]. Horizon Europe, the EU's primary research and innovation program, has allocated a budget of 94.30 billion USD for 2021–2027, with substantial funds earmarked for robotics-related projects [76]. In 2019, the UK government allocated approximately €34 million ($48 million) towards developing robotics for adult social care, showcasing the potential of robotics to transform care systems [29]. Japan's "New Robot Strategy" invested over 930.5 million USD in 2022 to support key sectors such as manufacturing, nursing, medical services, infrastructure, and agriculture [29]. China's "14th Five-Year Plan" for Robot Industry Development includes the "Intelligent Robots" program, which launched with an investment of 43.5 million USD [76].

3.2 The Importance of Computation Performance for Robotics

3.2.1 Robotics Computation is Becoming Increasingly Intensive

Robotics tasks are becoming increasingly compute-intensive due to several key trends and advancements in technology and application demands.

Complexity and Diversity of Tasks

Modern robotics are increasingly deployed in complex environments requiring high-level autonomy and adaptability. This complexity necessitates advanced algorithms that can process large volumes of data from various sensors in real time, significantly increasing computational needs [159].

Integration of Advanced AI and Machine Learning

The integration of deep learning and other AI technologies has revolutionized robotics, allowing for capabilities such as enhanced perception, decision-making, and language processing. These technologies, however, are computationally intensive, especially when processing real-time data streams or requiring continuous learning from the environment [149].

Increased Use of Vision and Sensory Data

Robots today often utilize sophisticated multi-sensor systems including vision, tactile, and auditory sensors that generate large amounts of data needing immediate processing. This multi-modal data acquisition and processing are critical for tasks like navigation, manipulation, and interaction, which demand significant computational resources [159].

Safety and Reliability Demands

For applications in critical sectors such as healthcare, automotive, or manufacturing, robots must meet stringent safety and reliability standards. Ensuring these standards often requires redundant systems and sophisticated fault detection and correction algorithms, which in turn increase the computational load [243].

3.2.2 Why is Computation Speed Crucial in Robotics?

Computation speed is critical for robots because it directly influences their ability to perform tasks efficiently and react to dynamic environments in real time. In robotics, every process from sensory data interpretation to motion planning and execution relies heavily on the underlying computational capabilities. Faster computation allows robots to process complex algorithms that enable them to perceive their surroundings, make decisions, and act on these

decisions swiftly. This is especially important in scenarios like autonomous driving, surgical robotics, or industrial automation where delays in processing could lead to errors or accidents. For example, in autonomous vehicles, computation speed affects the vehicle's ability to analyze and respond to sudden changes in its environment, such as detecting a pedestrian crossing or another vehicle suddenly braking.

The performance requirements vary by application and are specifically tailored to each (see §2.3). For example, in autonomous vehicles, the *latency*–the time from sensing an event (e.g., a change in distance, a new object) to the vehicle coming to a complete stop–requirement is calculated as follows:

$$(T_{computation} + T_{transmit} + T_{mechanical}) \times v + \frac{1}{2} \times a \times {T_{stop}}^2 \leq D \tag{3.1}$$

Here, $T_{computation}$ is the time taken by the computing system to process sensor inputs and issue control commands; $T_{transmit}$ is the transmission time of these commands to the vehicle's actuators; $T_{mechanical}$ is the reaction time of the vehicle's mechanical systems; v is the vehicle's speed; a represents the braking deceleration; D is the detected distance to the object; and $T_{stop} = \frac{v}{a}$ is the time for the vehicle to come to a full stop.

Given that $T_{transmit}$ and $T_{mechanical}$ are relatively minor (e.g., 1ms and 19ms, respectively, in [253]), the most significant factor is $T_{computation}$. Lower computation latencies allow the vehicle to maintain safety at higher speeds by reacting to farther objects. For instance, with a worst-case compute latency of 740ms as noted in [253], a vehicle can avoid obstacles detected at a minimum of 8.3m away when traveling at 5.6m/s.

In a similar vein for drones, the safe operational distance is calculated as follows:

$$D_{safe} = T_{reaction} \times v + \frac{1}{2} \times a \times T_{stop}^2 \tag{3.2}$$

In this equation, D_{safe} is the safe operational distance; v is the drone's velocity; a is the deceleration or velocity change due to control actions; $T_{reaction}$ encompasses the time from detection to actuation, including sensor processing, computation, and command transmission.

The drone's ability to respond to environmental changes is largely determined by its processing speed, which significantly influences its reaction time, denoted as $T_{reaction}$. Reducing the computation time, $T_{computation}$, can substantially increase the drone's maximum safe speed, thereby directly decreasing the overall mission time. For instance, assuming $T_{reaction} \approx T_{computation}$, doubling the computation speed could cut the mission time in half.

While latency is a critical metric, computation *throughput*–the number of control commands processed per second–also significantly impacts the quality of a robot's mission. In the context of autonomous vehicles, higher throughput allows for smoother vehicle control, minimizing abrupt maneuvers and braking [253].

3.2.3 The Evolution of Robotics Compute Platforms

Driven by the demands of applications, processors used in robotics have undergone significant evolution. Figure 3.2 encapsulates this development, illustrating the performance in Giga Floating Point Operations Per Second (GFLOPS) and Tera Operations Per Second (TOPS) for two distinct types of processors employed in robotics: Raspberry Pi processors, which are used for low-end, cost-effective robotics solutions, and NVIDIA Jetson processors, which cater to high-end robotics applications.[1]

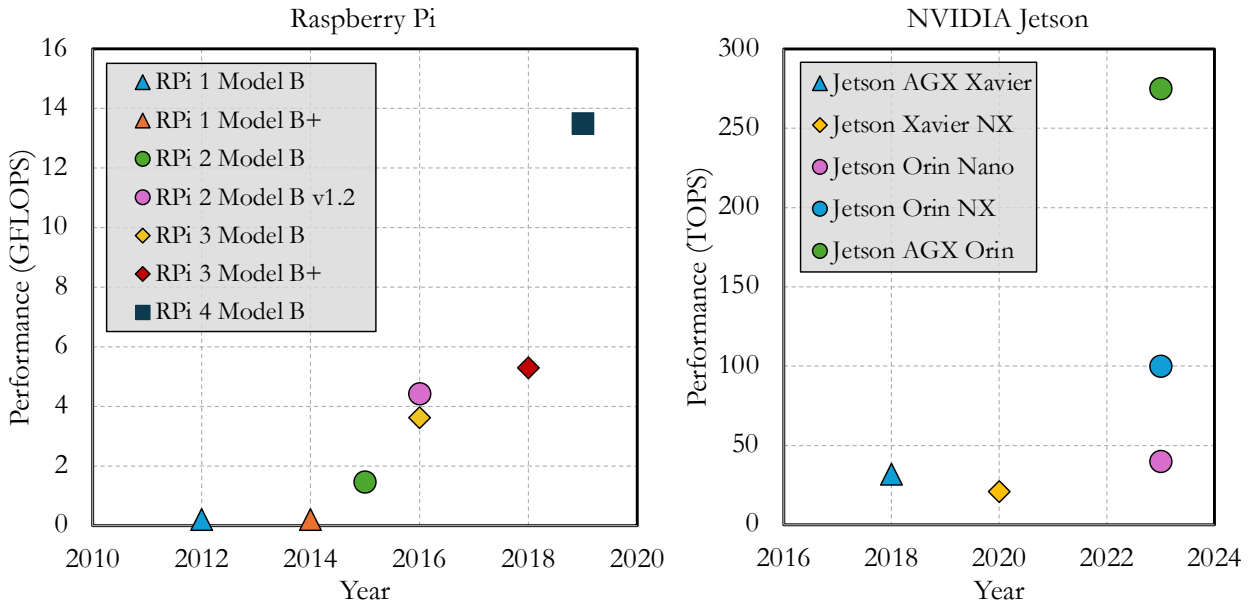


Figure 3.2: The evolution of low-end (left) and high-end (right) robotics processors over time.

Over time, these processors have progressively become more powerful, delivering higher performance levels. This pattern is consistent across robotics processors from various manufacturers, as further discussed below. This trend reflects the relentless demand for increased performance in the world of robotics.

The significant performance boosts in robotics processors across generations have stemmed from adopting increasingly aggressive architectures. For instance, the Raspberry Pi series [71] evolved from the Raspberry Pi 1, which featured a single-core ARM11 CPU and 256 MB of RAM, to become a staple in educational and hobbyist robotics projects. The Raspberry Pi 2 introduced a quad-core ARM Cortex-A7 CPU, significantly enhancing processing capabilities. This progression continued with the Raspberry Pi 3 and 4, with the latter offering a quad-core ARM Cortex-A72 CPU and up to 8 GB of RAM.

The NVIDIA Jetson series [46] represents a pivotal development in embedded AI and robotics processing. Starting with the Jetson TK1, which included a single-core Denver CPU and a Kepler-based GPU, NVIDIA began its foray into GPU computing for embedded systems. Subsequent models, the TX1 and TX2, incorporated the more powerful

[1]The use of different performance metrics and the omission of other platforms are due to the unavailability of public data.

Pascal GPU architecture, improving the number of CPU cores and energy efficiency. The series advanced significantly with the Xavier model, which introduced the Volta GPU architecture and Tensor Cores designed for AI acceleration, substantially increasing processing power. The latest in the series, the Jetson Orin, features the state-of-the-art Ampere architecture GPU and ARM Cortex-A78AE CPUs, marking significant advancements in AI performance and efficiency.

Intel's processor offerings in robotics have also evolved notably, with several series developed for diverse robotics applications. The Intel Atom series [32], initially aimed at low-power applications, has improved in performance-per-watt and integrated graphics, becoming more suitable for mobile robotics. The Intel Core series [36], particularly popular models like the i5 and i7, are used in high-demand robotics applications and have seen advancements in core counts, clock speeds, and AI integration, addressing the needs of complex robotic systems that require extensive computational resources. Intel Xeon processors, tailored for high-performance computing, offer significant cache sizes and advanced parallel computing capabilities essential for managing large data sets and conducting intensive robotics computations.

The Qualcomm Robotics RB platform [86] has rapidly adapted to the evolving needs of modern robotics with its RB3 and RB5 platforms. The RB3, powered by the Snapdragon 845 SoC, featured an octa-core CPU and the Adreno 630 GPU, balancing performance and power efficiency. The RB5 platform marked a significant advancement with the Qualcomm QRB5165 processor, utilizing the 7nm process technology for enhanced efficiency and performance. This processor includes an octa-core setup of Cortex-A77 and Cortex-A55 cores and the Adreno 650 GPU, alongside specialized AI hardware accelerators like the Hexagon 698 DSP with Tensor Accelerator, supporting up to 15 TOPS of AI performance.

3.3 The Thesis's Methodology

This thesis endeavors to bridge the fields of robotics and computer architecture, in part, by developing efficient computational architectures. This approach involves identifying and profiling compute-intensive tasks within robotics and mapping these tasks to various system-level architectural concepts, as depicted in Figure 3.3, to explore potential efficiency improvements.

The development of these architectural solutions adheres to several key principles detailed below.

3.3.1 Design Philosophy

Our architectural design philosophy posits that superior computer architecture enhances *(i)* data locality exploitation and *(ii)* the management of fine-grained parallelism.

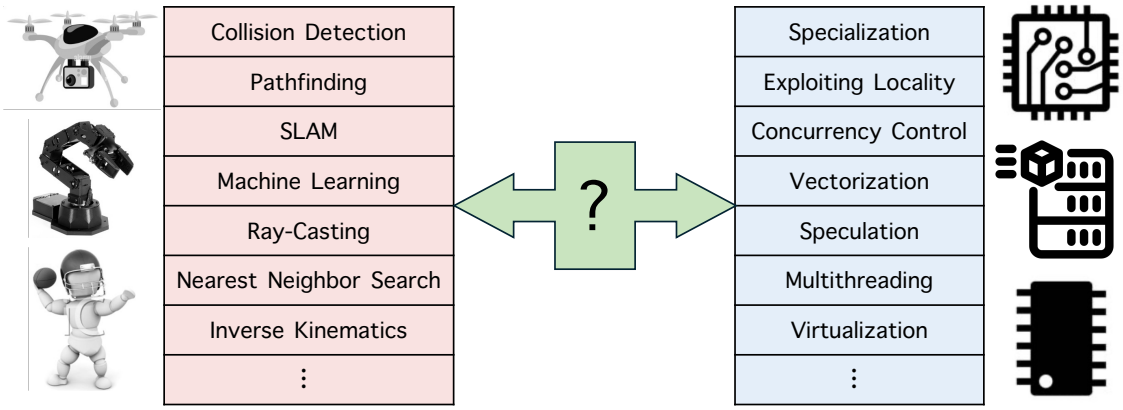


Figure 3.3: Mapping compute-intensive robotics operations to different architectural concepts. The mapping is illustrated abstractly in this figure; Chapters 6 and 7 elaborate on our methodology, specifically focusing on the identification of intensive kernels in existing and prospective robots, and their mapping to efficient architectural solutions.

(i) The concept of "locality" pertains to several forms of locality in real-world applications, such as data access and compute-data locality. Enhanced data access locality can be achieved through advanced caching techniques, including specialized caches (§6.2.1), cache prefetching (§7.6.4), and cache partitioning (§7.7.2). Compute-data locality benefits from optimized compute scheduling strategies, such as application-specific scheduling (§6.2.1) and system-aware scheduling (§6.2.2).

(ii) Fine-grained parallelism, inherent in many applications, encompasses concepts like pipelined processing and instruction-level parallelism. This type of parallelism is often *not* effectively exploited by software-level multithreading due to its high overheads [164]. Architecturally, however, it can be harnessed through specialized hardware accelerators (§6.2.1; §7.5.3), vectorization (§7.4; §7.6.3), and speculative parallelization (§6.2.2).

3.3.2 Practical Architectures

While specialized architectures can significantly enhance performance for specific tasks, a critical question remains: "how much specialization is too much specialization?" Given that hardware vendors are unlikely to produce a unique processor for each task, architectural designs must balance several factors to be considered practical. Primarily, *(i)* architectures should retain a degree of generality, and *(ii)* the performance benefits should justify the design costs. These factors are influenced by various non-architectural elements, including the anticipated use and significance of the application, potential market revenue, and the specific costs associated with developing new hardware.

A practical guideline is that *(i)* an architecture is deemed sufficiently general if it optimizes tasks that are critical and time-consuming across multiple important applications. For instance, the "collision detection" tasks targeted in this thesis are crucial, consuming up to 99% of processing time in diverse robotics applications such as autonomous vehicles, robotic arms, and humanoid robots (refer to §6 §7.4, and sources [123, 160, 182, 214, 226, 256]). *(ii)* An architecture meets the performance over cost criterion if the performance improvements significantly outweigh the

costs of design changes. This balance is explored through various design scenarios within this thesis, providing insights into the trade-offs between cost and performance (§6.4.6; §7.8.1; §7.8.2).

Chapter 4

RTRBench: Benchmarking Robotics Tasks

The development of hardware techniques to enhance the computational capabilities of robots is constrained by the absence of a comprehensive benchmark suite. In this chapter, we introduce *Real-Time Robotics Benchmark (RTRBench)*, a benchmark suite designed for robotic tasks. This suite encompasses a broad spectrum of tasks covering the entire software pipeline of most autonomous robots. *RTRBench* comprises 16 tasks across robot perception, planning, and control stages.

Diverging from most previous proposals that primarily utilize Python, our suite is developed entirely in C++ to ensure rapid execution. While Python modules, integral to earlier Python-based suites, have undergone extensive optimization, their performance lags behind that of implementations based on C++.

Crucially, for the evaluation of novel hardware techniques, tasks must be readily simulatable on micro-architectural simulators prior to any hardware fabrication. To facilitate this, we have designed a *harness* for every task, that interfaces with the simulator, managing the simulation process efficiently.

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 tasks implementing perception, planning, and control tasks. Many prior proposals (e.g., [82, 88, 162]) 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., [14, 221]) sacrifice performance for implementation ease.

3. **Easy-to-simulate:** *RTRBench* implements programs such that they are easy to simulate by current micro-architectural simulators (details in §4.3). Most prior proposals do not offer this feature; for example, the Python

runtime of [80, 162, 221], or the TCP-based communication primitives of [80] pose significant challenges to current simulators.

Furthermore, we investigate the architectural impacts of executing these benchmarks on a state-of-the-art robotic processor. This analysis identifies key sources of inefficiency within the architecture and proposes potential avenues for enhancement.

Publications & Materials

RTRBench is published in [116], and its source code is accessible at: https://github.com/cmu-roboarch/rtrbench

4.1 Analysis Framework

For simulation experiments, we use the ZSim [224] micro-architectural simulator and model a processor whose specifications resemble the Intel Core i3-8109U [105]. Intel Core i3-8109U is a state-of-the-art low-end processor used in robotic systems like the LoCoBot manipulator [103] that we will study in this chapter.

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

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

4.2 RTRBench Tasks

Table 4.1 summarizes *RTRBench*'s tasks along with their key characteristics. We select tasks 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* [158] 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 [179] are used for planning. We include both tasks in the suite to represent various real-world robots.

Following, we provide a description of our programs, along with their architecture-level evaluations. Noteworthy, while we evaluate the programs in the context of a simulation framework, they can be employed in scopes beyond simulation, including in ROS-like middlewares and real-world robots.

Table 4.1: RTRBench's tasks and their key characteristics.

Task	Stage	Bottleneck(s)	Task	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

4.2.1 pfl

Description: Monte Carlo localization [254], a.k.a. *particle filter localization* (PFL), is a method to estimate a robot's state (location, orientation) as it moves and senses the environment, *given a known map*.

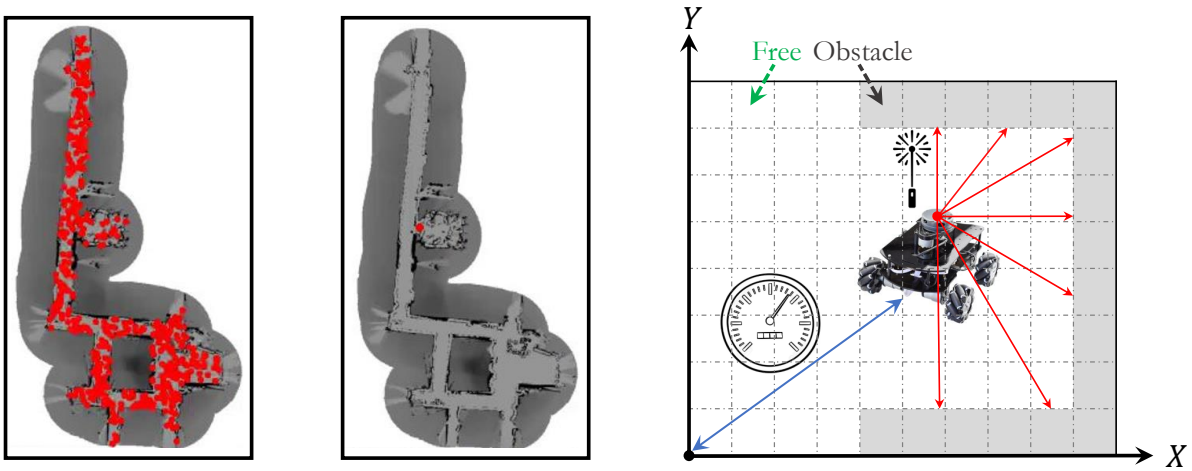
Figure 4.1 shows an overview of the program in an environment modeling a robot moving in CMU's Wean Hall [11]. The robot is equipped with an *odometer* and a *laser rangefinder*.

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 (Figure 4.1.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 (Figure 4.1.b).

The odometer measures the distance traveled by the robot at each step (the blue arrow in Figure 4.1.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 Figure 4.1.c). The laser readings are used to update particles' hypothesis of the obstacles' position. We evaluate the program 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 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 the architecture of the processor we propose in Chapter 7 (see §7.4).



(a) Particles in the beginning

(b) Particles after convergence

(c) Odometry (blue) and laser (red) measurements in the wild

Figure 4.1: Monte Carlo localization.

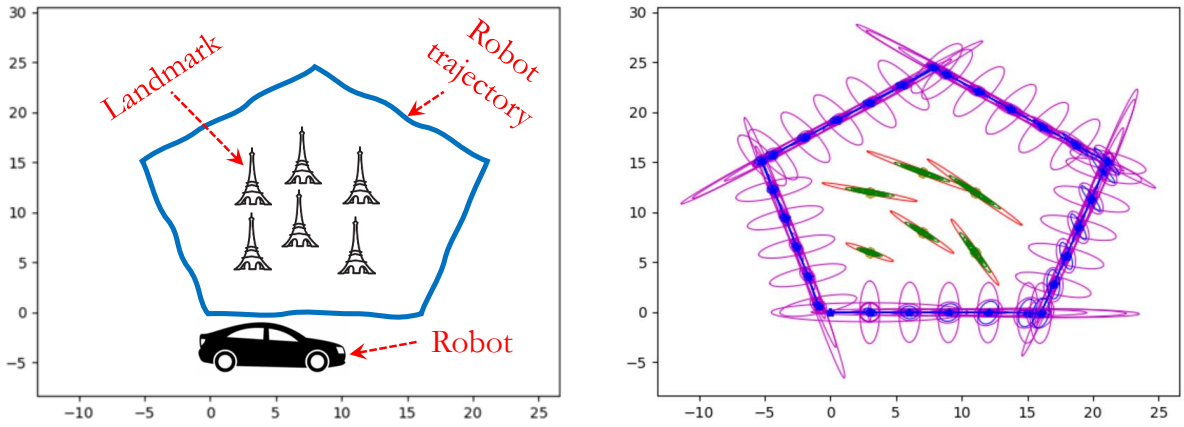
4.2.2 ekfslam

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 [240]. *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.

Figure 4.2 shows an overview of the program in an environment modeling a robot moving through a 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. Figure 4.2.a 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.

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 and fit in the caches (the size of matrices is proportionate to the *number of different measurement types*; *distance* and



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

Figure 4.2: SLAM using Extended Kalman Filter.

angle in the modeled application), parallel near-cache computation methods [207] seem a promising approach for performance improvement.

4.2.3 srec

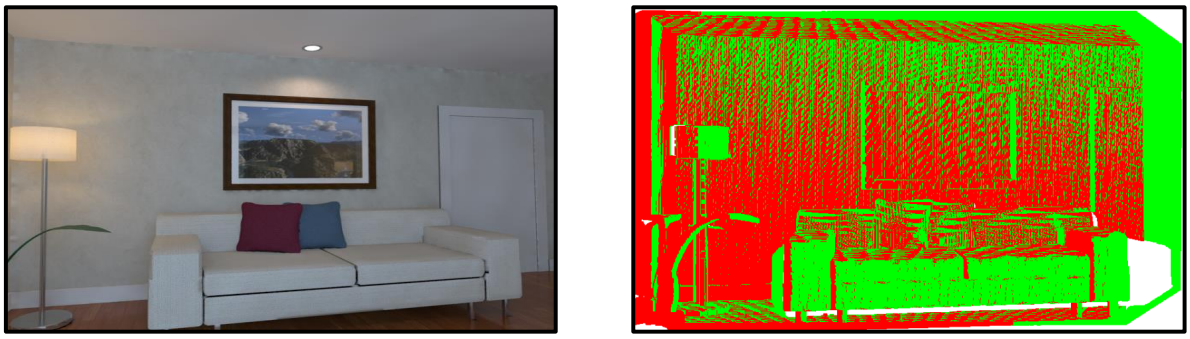
Description: Scene reconstruction [167] is the process of capturing the shape and appearance of the objects in an environment. We implement the scene reconstruction mechanism of [167], a real-time 3D reconstruction mechanism in *dynamic* scenes. It uses the *iterative closest point (ICP)* algorithm of prior work [205] 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 [167], 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 program using the `living_room` inputset from the ICL-NUIM [157] dataset. Figure 4.3.a shows the environment (one photo out of all taken by the robot's camera), and Figure 4.3.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. Near-data processing approaches [201] seem more fitting, particularly because of the low compute-to-communicate ratio of data.

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.



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

Figure 4.3: 3D reconstruction in dynamic scenes.

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.

4.2.4 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 (Figure 4.4.a), and the planner *searches* it using a graph search algorithm. A* [158], along with its variants and extensions, is the seminal algorithm widely used in various robot path planning applications. The key novelty of A* 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 (Figure 4.4.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 [237], which is a snapshot of Boston, Massachusetts, as the environment (Figure 4.4.b). The car's length$\times$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, 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

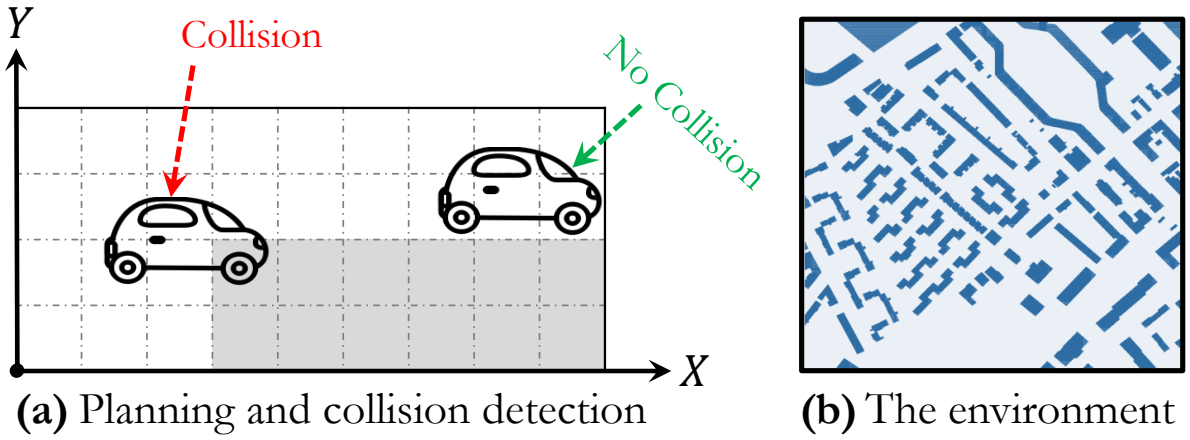


Figure 4.4: 2D path planning.

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.

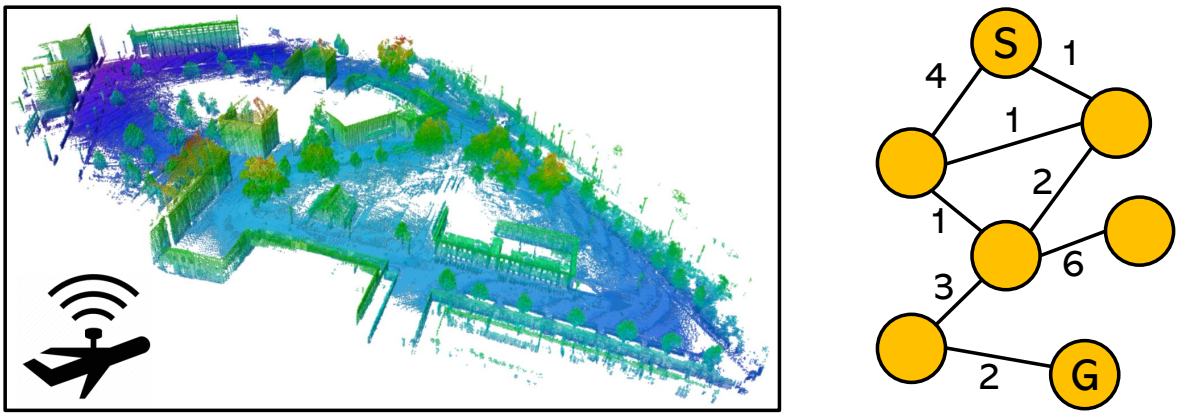
Significant fine-grained parallelism and spatial locality make hardware acceleration a perfect fit for collision detection.

4.2.5 pp3d

Description: We implement a mobile robot navigating in a 3D environment. The task 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 [23], which is a snapshot of Freiburg campus (Figure 4.5.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.

Evaluation: Other than collision detection, the graph search is another major performance bottleneck. Figure 4.5.b shows an example of the graph search problem. Search algorithms like Dijkstra and A* try to find the shortest path between a start point (e.g., 'S' in Figure 4.5.b) and a goal point (e.g., 'G' in Figure 4.5.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 [229] and found that it can eliminate around one-third of the data misses. Also, *speculative parallelism* approaches [164] could be quite effective in parallelizing such hard to parallelize graph search algorithms.



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

Figure 4.5: 3D path planning.

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 [187], could significantly accelerate the search process.

4.2.6 movtar

Description: This task represents a complex planning problem, in which a robot is trying to catch a *moving* target (Figure 4.6). 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* [120] 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** (WA*) [216] instead of A* 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.
Evaluation: The performance of the task is largely dependent on the inputset. In large environments, the task 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* [143, 189] can be used for improving the performance of heuristic calculations. Heuristic values, particularly when tight *optimality guarantees* are not required, are amenable for approximation.

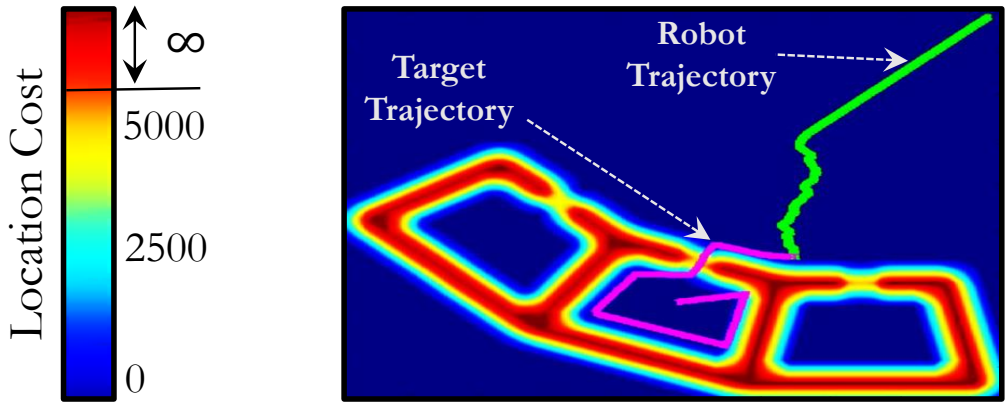
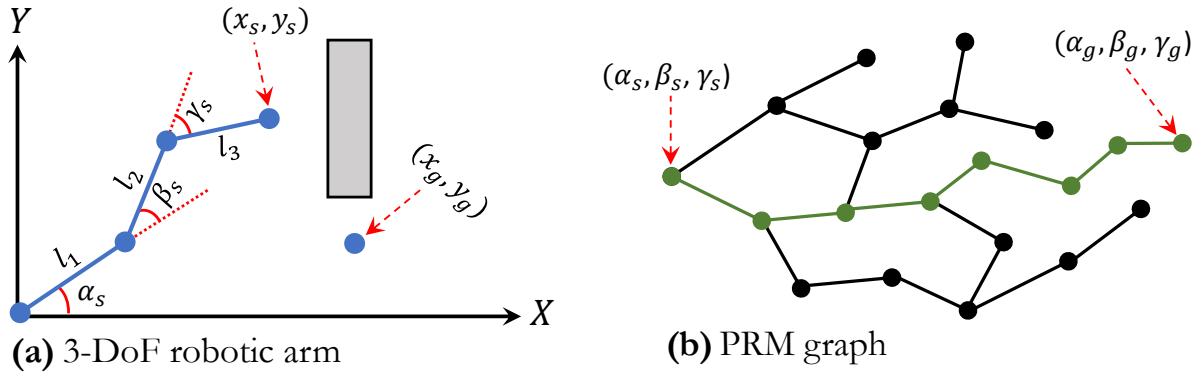


Figure 4.6: Catching a moving target.

4.2.7 prm

Description: Motion planning for stationary robotic arm manipulators with multiple degrees-of-freedom (DoF) is one of the challenging, time-consuming tasks in robotics. The problem has been targeted in a variety of levels from algorithm to, recently, architecture [182, 200].

Figure 4.7.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

Figure 4.7: 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)* [166] 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 Figure 4.7.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* to find a path from the start to the goal (green path in Figure 4.7.b). We model a 5-DoF arm manipulator operating in two synthetic environments, as shown in Figure 4.8. `Map-F` represents a free environment, and `Map-C` shows a cluttered one.

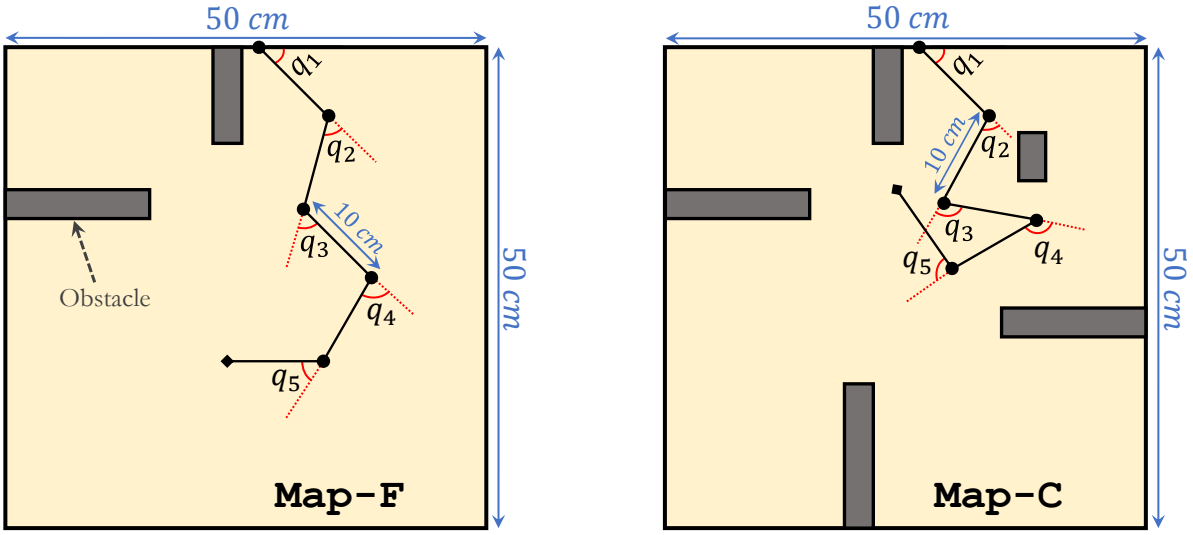


Figure 4.8: Synthetic maps for evaluating the robotic arm.

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.

4.2.8 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) [179] 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 Figure 4.9. During extending the tree, RRT checks the collision status of different configurations with the obstacles in the environment. We evaluate the task

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.

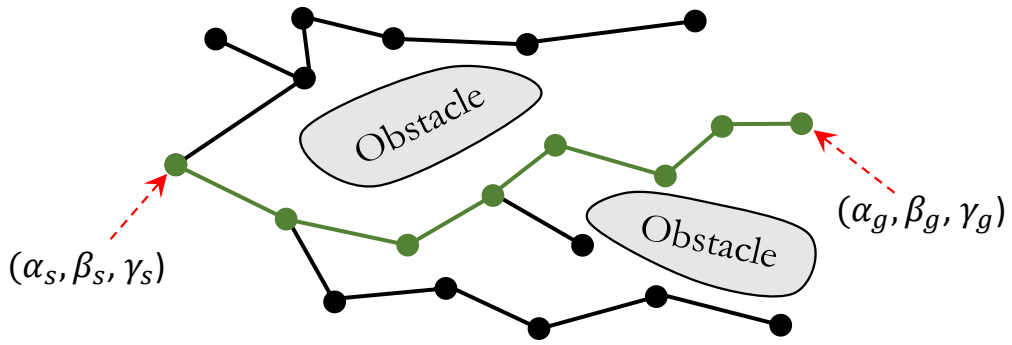


Figure 4.9: Arm manipulator planning by the RRT algorithm.

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 [160, 182, 226].

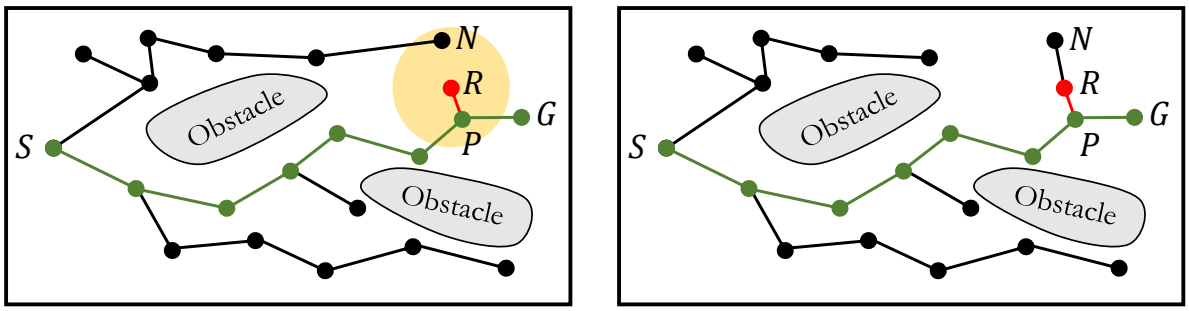
Nearest neighbor search is another performance bottleneck, taking up to 31% of the execution time. When drawing a sample, RRT searches the other samples to find the near ones to connect the new sample to them. This operation exhibits irregular memory accesses, because the samples whose values (angles) 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.

4.2.9 rrtstar

Description: RRT is fast but can return an inefficient, costly path. RRT* [152] 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.

Figure 4.10 shows an example of rewiring. Figure 4.10.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 Figure 4.10.b.

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



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

Figure 4.10: A rewiring example with RRT*.

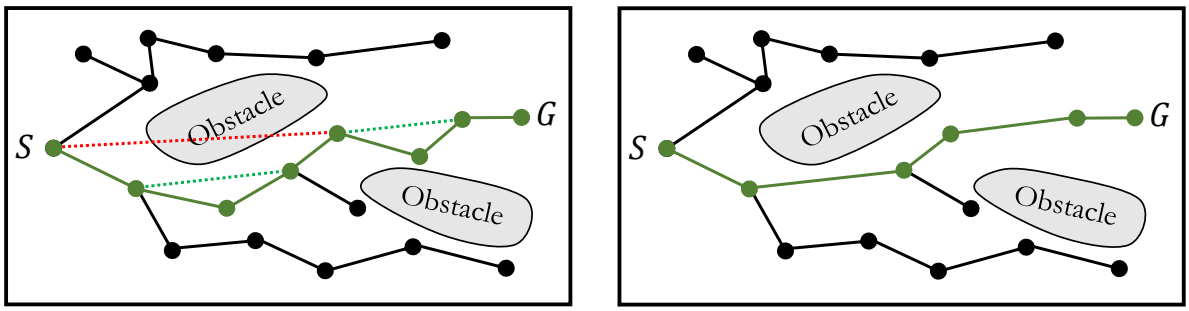
Evaluation: RRT* is significantly slower (up to 8× in our experiments) but generates shorter paths (1.6× on average) as compared to RRT.

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.

4.2.10 rrtpp

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

Prior work [148] 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. Figure 4.11 shows examples of shortcutting.



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

Figure 4.11: Post-processing the path found by RRT.

Figure 4.11.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 Figure 4.11.a, the two node pairs connected by dashed green lines can be shortcutted, while the node pair connected by a dashed red line cannot. Figure 4.11.b shows the path after post-processing. The

post-processing step could run for several iterations to further reduce the path cost.

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.

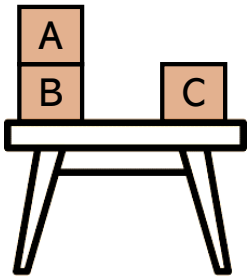
4.2.11 sym-blkw

Description: Symbolic planning [122] 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 [156] in its context. Figure 4.12.a shows parts of a symbolic representation of the blocks world problem, and Figure 4.12.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 [235].

```
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

Figure 4.12: Blocks world problem.

Evaluation: The task has only two 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 programs 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 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 [128]. Such accelerators can be repurposed for accelerating symbolic planning, with minimum effort.

4.2.12 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 [104]. 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. Figure 4.13 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)
        ⋮
```

Figure 4.13: Firefighter robots.

Evaluation: The program 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.

4.2.13 dmp

Description: *Dynamic movement primitives (DMP)* [175] is a control task 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* [161] 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 Figure 4.14. The black lines in Figure 4.14 show the trajectory (left) and velocity (right) generated by *DMP*.

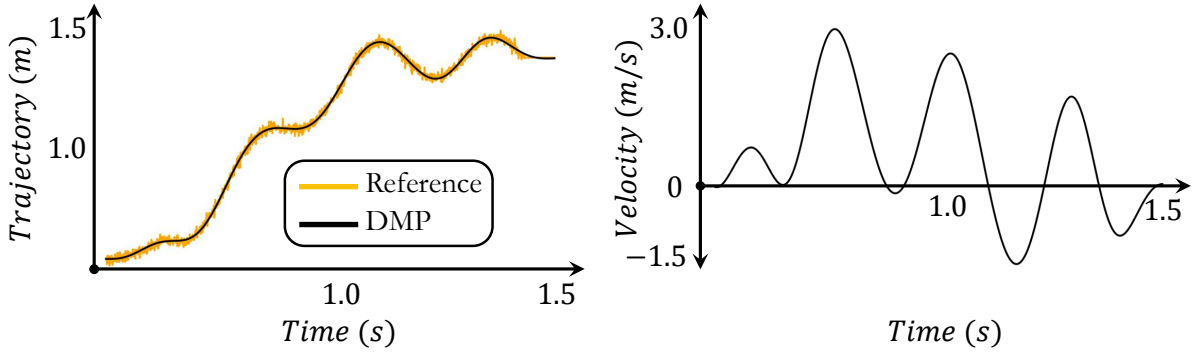


Figure 4.14: Dynamic movement primitives.

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 [208] have been shown to be effective for this kind of computation.

4.2.14 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 [144]. 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.

Figure 4.15 shows an overview of the program 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.

Evaluation: The major bottleneck of the task is solving the optimization problem, taking more than 80% of the entire execution time. RoBoX [220] proposes a full-fledged accelerator for accelerating this process. It uses specialized logic and near-data computation to solve the problem significantly faster.

4.2.15 cem

Description: We model a ball-throwing robot whose throwing *skills* get improved using *reinforcement learning*. We use *V-REP* [219] to simulate the robot and the environment. Figure 4.16 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

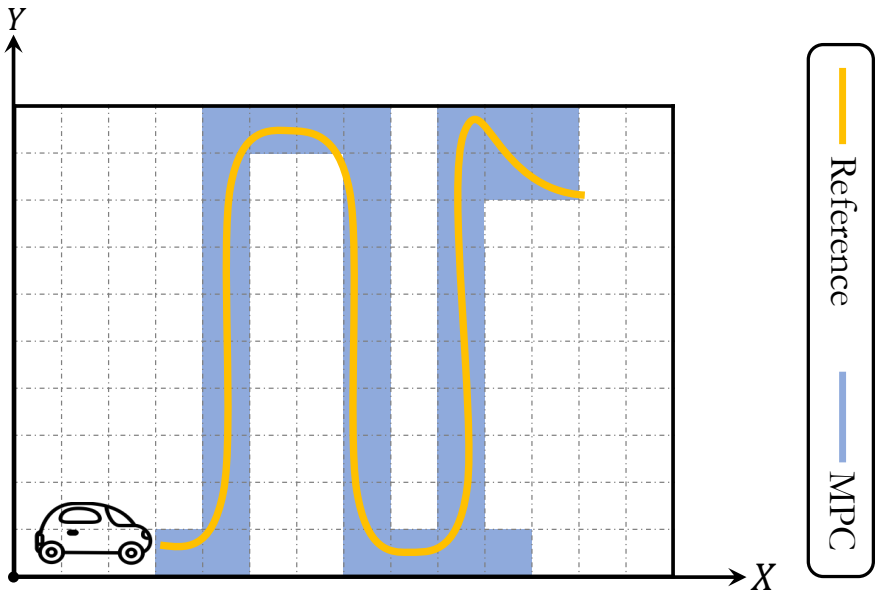


Figure 4.15: Model predictive control.

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.

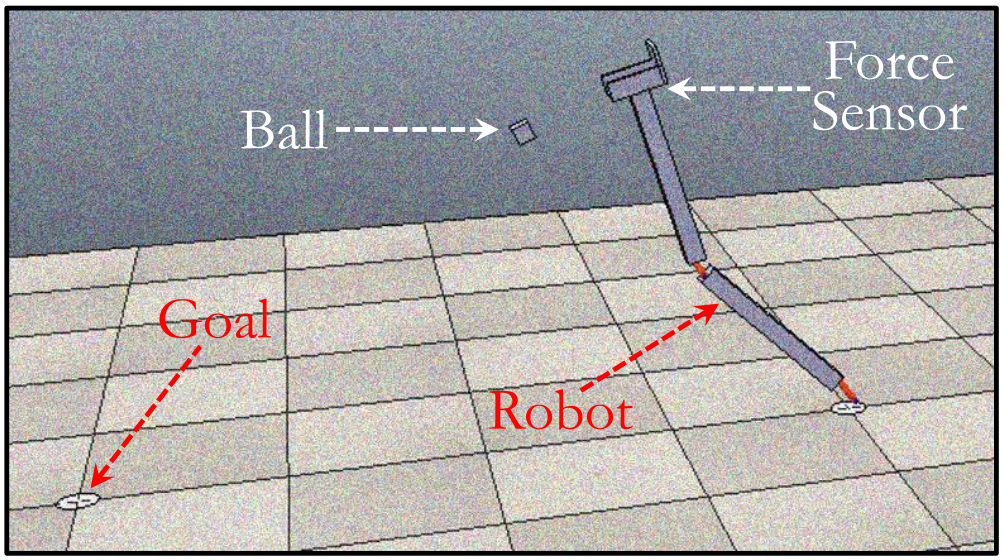


Figure 4.16: Ball-throwing robot.

Cross-entropy method (CEM) [212] 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. Figure 4.17 shows how reward (higher is better) changes over learning.

Evaluation: Recent work [227] proposes hardware acceleration for reinforcement learning that can be well applicable

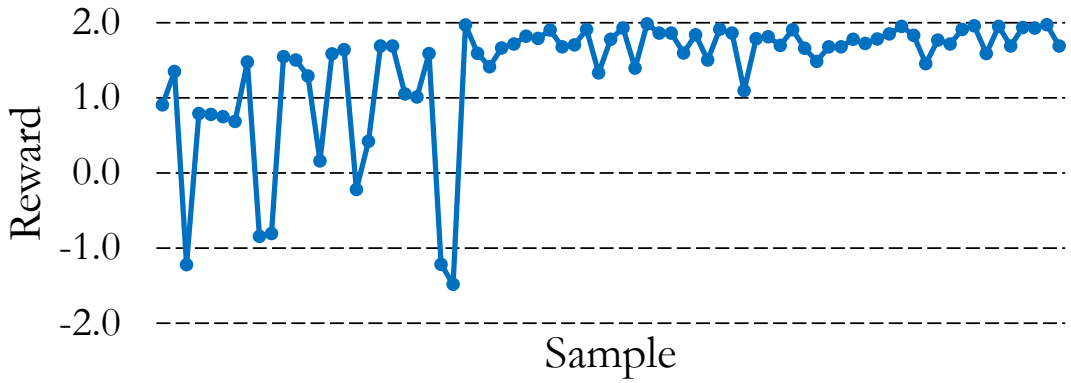


Figure 4.17: Rewards over time using CEM.

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.

4.2.16 bo

Description: In robotics, *Bayesian optimization (BO)* is used to optimize control parameters in reinforcement learning [163]. *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. Figure 4.18 shows how the reward changes over the course of the 45 iterations of the learning process.

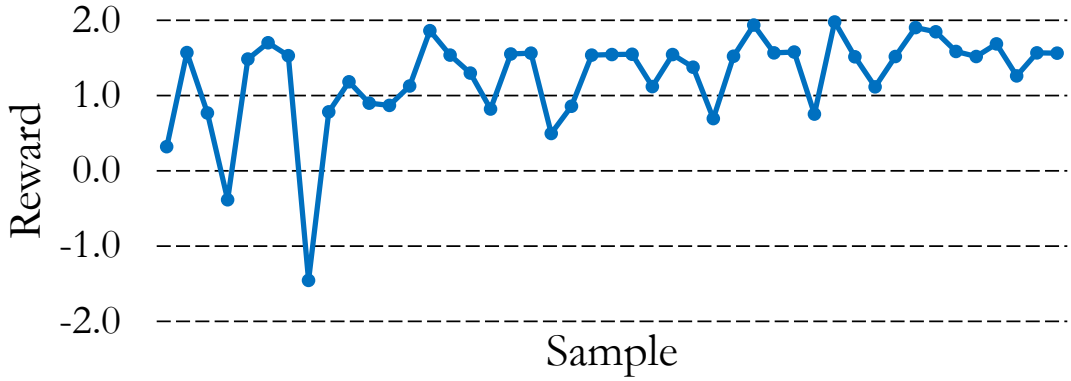


Figure 4.18: Rewards over time using BO.

Evaluation: The task exhibits largely the same characteristics as cem. However, computationally, it is more intensive ($\sim$15000$\times$ more iterations). Hardware acceleration of the reinforcement learning program can be a perfect architectural solution to accelerate the application. Also, since more metadata is kept with *BO*, its sort operation is more time-consuming ($\sim$6$\times$ as compared to cem).

4.3 Implementation Details

System Requirements: We test *RTRBench* under Ubuntu 18.04 with Linux Kernel 4.15, and compile with GCC 11. 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 (Clang: 5; Visual Studio: 15.8).

Also, we integrate the programs with ZSim [224], 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 usage help message for every program. Running the binary file of a program with `--help` will print the help message. For example, Figure 4.19 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
```

Figure 4.19: An example of a help message.

Also, as the figure shows, we implement the programs 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 this chapter, we report execution results for one inputset per task. However, in the repository, we provide multiple inputsets for many of the tasks.

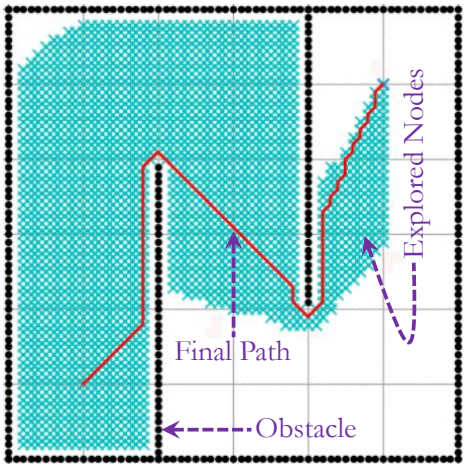
4.4 Comparison with Open-Source Repositories

There are a few educational open-source libraries that provide implementations of robotic tasks. 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* [66,221] (denoted as *P-Rob*) and *CppRobotics* [14] (denoted as *C-Rob*). *P-Rob* is a popular robotic library with $\sim$6.3K forks and $\sim$20.6K stars (as of May 2, 2024). *P-Rob* provides a Python code collection of the robotic tasks operating on small, synthetic inputsets. *C-Rob* translates some of the *P-Rob* programs to C++.

As a case study, we compare pp2d with the counterpart programs 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 [102] cores operating at 2.60 GHz, with the operating system and compiler described in §4.3.

As inputset, we use the map provided by the competitor libraries, which is depicted in Figure 4.20.a. Because the map is small, we also scale it by different factors to evaluate the implementations in larger (or finer-resolution) environments. Figure 4.20.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

Figure 4.20: 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 task, 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.

4.5 Chapter Summary

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

Chapter 5

RoWild: Architectural Implications of Robotics on Modern Hardware

This chapter introduces *RoWild*, a comprehensive, open-source, cross-platform robotic benchmark suite. *RoWild* builds upon *RTRBench*, modeling end-to-end robotic applications and supporting multi-platform execution.

Benchmarking robotics poses a significant challenge due to the diversity of applications, ranging from self-driving cars to home-assistant robots, with future possibilities continually expanding. It is impractical to represent all these applications within a single benchmark suite. *RoWild* addresses this challenge by recognizing that various robots, despite their different applications, perform a finite set of common "robotic tasks" such as scene understanding and pathfinding. For instance, both self-driving cars and home-assistant robots require scene understanding. However, the algorithms and constraints for these tasks can vary considerably across different applications.

RoWild includes a broad spectrum of individual robotic tasks, covering the software pipeline of most autonomous robots. Designed with versatility in mind, *RoWild* implements each task using various algorithms and parameters, allowing for flexible task configuration and pipelining to simulate the end-to-end computation of diverse robotic applications. This approach enables *RoWild* to model a wide range of robotic applications, providing a comprehensive benchmark suite.

The selection of tasks and algorithms in *RoWild* is informed by an analysis of 29 industrial robots, encompassing a diverse range from arm manipulators and home-cleaning robots to self-driving vehicles. This analysis ensures a broad and relevant task selection. Additionally, *RoWild* integrates state-of-the-art research algorithms, such as deep learning-based pathfinding, positioning it as a forward-looking benchmark suite for robotics. Distinguished from previous work, *RoWild* is high-performance, simulator-friendly, versatile, and modular.

The second contribution of *RoWild* is the investigation of architectural implications in robotics. Using tasks from *RoWild*, we model six different end-to-end robotic applications and evaluate them on various platforms, from low-

end embedded CPUs to high-end server-grade GPUs. Previous studies [124, 184, 253] have performed system-level analyses, such as CPU versus GPU comparisons, but these remained at a high level. Our study delves into low-level implications, examining aspects like caching, prefetching, and vectorization effectiveness.

Our system-level investigations reveal significant inefficiencies in the architecture of current computing platforms when executing robotic workloads. Specifically, we make the following observations.

- **Vectorization Is Ineffective.** Despite occupying substantial silicon area, CPU vectorization offers limited benefits for robotic tasks. This is particularly true in cases of axis-unaligned orientation, where the robot's memory layout is not conducive to vectorization.
- **On-Edge Parallelism Hits The Memory Wall.** Massive parallelism on edge platforms, such as NVIDIA's Jetson Nano, is hindered by memory constraints. The memory bottleneck occurs well before the limitations posed by Amdahl's law.
- **Simple Prefetchers Are Inadequate.** While basic data prefetchers can assist with robotics workloads, the complex AI algorithms used in robotics often employ irregular data structures. These structures result in memory access patterns that overwhelm commercial prefetchers.
- **Caches Perform Significant Unnecessary Data Movements.** Hardware caches, not being attuned to the software semantics of robots, operate counter-productively. As a result, they execute excessive data movements, leading to inefficient use of the memory hierarchy.

Publications & Materials

RoWild is published in [113, 114], and its source code is available at: https://github.com/cmu-roboarch/rowild

5.1 Existing Work

Table 5.1 puts *RoWild* in the context of existing work.

Several prior works [124, 184, 253] report the characteristics of a *particular* application of robotics. For instance, Lin et al. [184] and Yu et al. [253] study autonomous driving, and MAVBench [124] studies micro aerial vehicles. Each of these studies targets a particular application of robotics, leaving out many others (e.g., stationary arms, humanoid robots).

Similarly, prior work proposes benchmark suites [88, 162] and studies the efficacy of different algorithms for *particular* robotic tasks. For instance, *OMPL* [88] is a suite for sampling-based planning, and *RLBench* [162] is a suite for robot learning. Each of these suites targets a specific task, not the entire robotic software pipeline. Also,

Table 5.1: Feature comparison of related work and RoWild. More ✔ is better.

Paper/Repository	Scope	End-to-End	High-Perf.	Multi-Platform	Open-Source	Simulator-Friendly	Versatile & Modular	System-Level Analysis
Lin et al. [184] Yu et al. [253]	**Self-Driving Cars**	✔	✔✔	✔✔	✘	Unknown	Unknown	✔
MAVBench [124]	**Drones**	✔	✔	✔	✔	✣	✘	✔
One-off [88, 162]	**Single Task**	✘	✣	✘	✔	✣	✔	✘
ROS [80]	**Broad**	✘	✣	✘	✔	✘	✔	✘
Educational [14, 66]	**Broad**	✘	✘	✘	✔	✣	✘	✘
***RTRBench* (§4)**	**Broad**	✘	✔	✘	✔	✔	✔	✘
RoWild	**Broad**	✔	✔✔	✔✔	✔	✔	✔	✔✔

✣ Depending on the case (e.g., kernel, simulator), it can be ✘ or ✔. MAVBench [124] employs Unreal Engine to model the drone's surrounding environment. Consequently, the "simulator-friendly" feature of MAVBench is contingent upon the engine used. For instance, AirSim [15], with some engineering effort, can be simulated in many existing architectural simulators. Conversely, Unity [97], primarily developed in C#, cannot be simulated in numerous existing simulators. Both "high-performance" and "simulator-friendly" features of one-off benchmark suites [88, 162] are tied to the programming language utilized. RLBench [162], developed entirely in Python, is neither high-performance nor simulator-friendly. On the other hand, OMPL [88], developed in C++, boasts both high-performance and simulator-friendly attributes. The "high-performance" feature of ROS [80] relies on the kernel language. Some of ROS's kernels, developed in C++, are optimized for performance. Others are either written in Python, or written in C++ but lack optimization for performance. The "simulator-friendly" feature of educational repositories such as [14,66] depends on the language in which they are developed. For example, [66] is developed in Python and is not simulator-friendly. On the other hand, [14] is developed in C++ and can be simulated in today's architectural simulators.

combining these suites into a broad set of workloads is not straightforward, because they use unalike setups (e.g., C++ vs. Python).

ROS [80] is a robotic middleware, offering codes for package management, device control, and software libraries. Nonetheless, *ROS* does not consider performance as the main objective: over three-quarters of the codes are written in Python, and even those written in C++ are not all high-performance. Further, the use of Python and TCP-based inter-process communication primitives in *ROS* conflicts with many microarchitectural simulators, such as [131].

Educational repositories, such as [14, 66], implement various robotic tasks. Nevertheless, the performance of these repositories are orders of magnitude far from real-time, as show in §4.4. Our work *RTRBench*, presented in Chapter 4, featured some improvements in performance over previous suites; however, it is single-threaded and lacks essential features, such as end-to-end application modeling, cross-platform compatibility, and detailed system-level evaluations. *RoWild* extends *RTRBench*, rectifying these limitations and establishing itself as an apt benchmark suite for exploring the nexus between robotics and systems.

5.2 The RoWild Approach

5.2.1 Workloads

To model a variety of robotic applications, *RoWild* implements a rich set of individual robotic tasks and algorithms. Table 5.2 lists these tasks and the algorithms implemented to execute them, with symbols indicating the real-world robots that either come pre-packaged with the software where the algorithm is used or are custom-programmed in the literature to use the algorithm to perform a specific mission. In some cases, the algorithms are identified based on

the demystification of other works. However, it should not be assumed that the robots listed are certainly/exclusively using these algorithms.

Stage	Task	Algorithm(s) in RoWild
Perception	State Estimation	MCL[†‡ϒ¶ησ] [254], AMCL[ζ] [239]
	Localization and Mapping	EKF[ϒ§τφΛ] [240], Fast[‡¶ζ] [195], Graph[ϑφ] [225], ORB[ςρ] [129]
	Scene Reconstruction	Point-Based Fusion[†‡ϒκυ] [249]
	Grid Generation	Probabilistic Occupancy Map[†κϑ] [151]
	Object Detection	MobileNet-SSD[‡ϒρχΛ] [252]
Planning	2D/3D Navigation	WA⋆[†¶ζηαΠΛ] [216], GA⋆ [256], RA⋆ [118], IDA⋆[ς] [174] , DQN [194]
	Timed Search	WA⋆[†¶ζηαΠΛ] [216] + Backward Dijkstra [120]
	2D/3D *n*-DoF Search	RRT[†‡ϒκ] [148], RRT⋆[ϒωγδ] [152], PRM[ϒϑψ] [234], Shortcut [153]
	Reactive Planning	Behavior Tree[†λϑφ] [154]
	Task Scheduling	Symbolic Planning[§ϑδ] [122]
Ctrl.	Motion Control	MPC[ϒΠϑ†καψε] [144] PID[ϒϑζθφ] [247], PP[‡ϒκφ] [209]
	Parameter Learning	DMP[ϒϑξαθι] [175], CEM[§λ] [212], BO[ρ] [163]

Real-World Robots:

[†]Spot [9]
[‡]DJI Phantom [61]
[ϒ]LoCoBot [103]
[§]Atlas [8]
[¶]AMR [72]
[Π]AscTec Pelican [6]
[ζ]Roomba 980 [78]
[υ]Roomba i7+ [79]
[η]Husky [26]
[ϑ]YuMi [101]
[κ]Jackal [44]
[ξ]LBR [52]
[ρ]Pepper [85]
[σ]TurtleBot [94]
[ς]TurtleBot3 [95]
[τ]MiR [54]
[φ]AG [50]
[Λ]Pioneer 3-DX [64]
[φ]UR10e [98]
[χ]TALOS [87]
[ψ]KR60-3 [51]
[ω]Grizzly [12]
[γ]Skydio [84]
[δ]M-200iA/2300 [20]
[ε]PerceptIn [252]
[α]Boxbot [10]
[θ]UR5e [99]
[ι]Franka Panda [22]
[λ]PAL TIAGo [92]

Table 5.2: RoWild's workloads.

Our selection of algorithms is the result of an exhaustive examination of the 29 real-world robots listed in Table 5.2 (symbols), ensuring that they represent a broad majority of robots employed in practical scenarios. We incorporate algorithms that *(i)* are applicable to real-time environments, *(ii)* have proven efficacy within the robotics community, and *(iii)* enable *RoWild* to emulate a wide and diverse array of end-to-end robotic applications (see §5.3).

Importantly, while *RoWild* emphasizes state-of-the-art algorithms, it also includes seminal algorithms that have stood the test of time and remain highly relevant in robotics. This is aligned with the interests of system manufacturers who strive to accelerate these algorithms. For example, [136], [171], and [137] fabricate hardware accelerators for the classic EKF [239], IDA⋆ [174], and RRT algorithms [179], respectively.

RoWild uses these tasks to model various robotic applications. For each robotic application modeled, *RoWild* selects the necessary tasks, *specializes* each task to suit the application's requirements (see §5.2.2), and integrates them into a pipeline to model the end-to-end functionality of the robot.

5.2.2 Key Features and Considerations

Our development of *RoWild* involves essential considerations that make it suitable for systems and hardware research, as outlined below:

Comprehensive: As discussed in §5.2.1, *RoWild* includes a diverse range of robotic workloads. This feature is crucial from a computer architecture perspective; it is unlikely that hardware vendors will fabricate a specific processor for

every robotic application.

High-Performance: Performance is at the heart of *RoWild*'s design, setting it apart from other open-source robotic repositories that face issues like Python overhead [80, 162, 221] and a lack of modern software techniques [14].

To optimize performance, we develop codes from scratch in native languages and use industry-standard profilers [31, 58] to identify execution bottlenecks and focus on accelerating them.

We make use of various high-performance software techniques including `constexpr` branches and `constexpr` functions that enable the compiler to perform calculations at compile time, thereby improving execution by eliminating not-taken branches. We employ built-in functions provided by GCC for aligning and prefetching data, providing branch prediction hints, and maximizing loop unrolling to enable efficient out-of-order execution. Additionally, we make use of the high-performance VCL [108] for manual code vectorization.

Simulator-Friendly: Because most hardware research uses simulators, we develop *RoWild* to be compatible with existing architectural simulators (e.g., no Python runtime, no TCP-based inter-process communications as in *ROS*). By default, *RoWild* codes are integrated with ZSim [224].

Cross-Platform: *RoWild* caters to a wide range of compute resources by developing applications in C++, CUDA, and Verilog. This is particularly relevant for modern AI/robotic platforms that come equipped with heterogeneous compute resources [57, 60].

Versatile: *RoWild* is a versatile repository that not only comprises diverse robotic workloads but also develops each of them in a configurable manner. This configurability empowers users to perform each task using a range of algorithms and parameters, enabling the study of realistic robots that feature specialized software components for specific robotic applications and environments.

Modular: *RoWild* embodies modularity in its design, enabling the seamless pipelining of different tasks to model end-to-end robotic applications. In each of the applications exemplified in §5.3, the integration of the various tasks into the pipeline required fewer than 20 lines of C++ code.

Open-Source: *RoWild* is distributed under the MIT License and is available as open-source software.

Flexible & Easy to Use: Similar to the approach in *RTRBench* (refer to §4.3), we include a *harness* and a help message for each program. Executing the binary file of a program with `--help` displays the help message. For instance, Figure 5.1 illustrates a sample help message.

5.2.3 Compute Platforms

Various compute platforms can be used in robotics. For example, safety-critical self-driving cars might use server-grade, high-end GPUs; on the other hand, low-cost surveillance robots might use affordable, low-end CPUs. *RoWild* evaluates a range of processors, as shown in Table 5.3 and detailed below:

```
$ ./ patrolbot.out --help

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

OPTIONS:
  --cfg <val>         Model configuration file
  --model <val>       Pre-trained model containing weights
  --labels <val>      Input file containing class names
  --imgdir <val>      Input images
  --log <val>         Input sensor log file
  --confidence <val>  Minimum confidence threshold for bounding boxes
  --scale <val>       Image scale factor
  --path <val>        Input path file
  --output <val>      Output log file

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

Figure 5.1: An example of a help message.

***LC*:** A low-end, quad-core ARM Cortex A57 CPU available on NVIDIA's Jetson Nano board [47]. It uses a 4GB LPDDR4 memory with up to 25.6GB/s bandwidth.

***LG*:** A low-end, 128-core GPU with Maxwell architecture available on NVIDIA's Jetson Nano board [47]. It shares the 4GB LPDDR4 memory system with the ARM CPU.

***HC*:** A high-end, server-grade Intel Xeon Gold 5218R CPU [41] with 20 two-way cores. It uses six 64GB DDR4 memory modules, each with up to 38GB/s bandwidth.

***HG*:** A high-end, super-clocked NVIDIA Titan X GPU [17] with 3072 CUDA cores featuring the Pascal architecture. It has a 12GB GDDR5 memory with up to 336GB/s bandwidth.

Table 5.3: The evaluated compute platforms (January 2024 prices).

Acronym	Platform	Cores	Freq. (GHz)	TDP (W)	Memory (GB)	Price (US $)
LC	ARM Cortex A57 CPU	4	1.43	10	4	149
LG	NVIDIA Maxwell GPU	128	0.92	10	4	149
HC	Intel Xeon Gold CPU	20 ($\times$2)	2.10	125	384	1493
HG	NVIDIA Titan X GPU	3584	1.41	250	12	999

5.2.4 Software Workflow and Validation

We develop CPU codes with C++17. We compile programs with GCC 11 with `-O3` and run them under Ubuntu 22.04. We develop GPU codes with CUDA 11 and compile them with NVCC. We set the number of thread blocks and threads-per-block empirically to obtain the fastest execution time for every application.

To model the producer-consumer execution model (i.e., pipelining the tasks), we use C++'s asynchronous execution primitives. Tasks run on separate threads (host or device), and producer-consumer pairs communicate through shared `future` objects. This way, different tasks are effectively pipelined, maximizing resource utilization.

Sanity-Checking

We conduct thorough sanity checks on *RoWild*. To ensure the validity of our end-to-end results, we compare them with outputs from other sources such as repositories [14, 66, 82] and mathematical benchmarks, such as verifying that the final path is indeed the shortest. *RoWild* supports two operational modes: *sanity-check mode* and *high-performance mode*. In sanity-check mode, the system integrates comprehensive, fine-grained assertions to verify correctness. This functionality is crucial for future researchers who may modify the software, as it helps prevent the introduction of errors. In contrast, high-performance mode omits these assertions to enhance execution speed.

Performance Validation

Because many factors (e.g., inputset, environment, configuration) affect the robotic workloads' performance, we refrain from making exact performance comparisons with other suites, and leave such task to potential users. Nevertheless, our measurements of *RoWild* (as detailed in §5.3) show that it is 1–3 orders of magnitude faster than the reported performance of *open-source* repositories like PythonRobotics [66], CppRobotics [14], and ROS [80]. These repositories suffer from the high overheads of Python runtime [66, 162], network communications (publisher-subscriber communication model in ROS [80]), and unoptimized software (e.g., single-threaded code [14, 66, 80, 162], spinner callbacks [80], poor memory management [14]).

Compared to *open-source* real-world robots (e.g., [103]), *RoWild* offers significantly superior performance. Such robots typically are developed with objectives such as ease of programming in mind, and frequently utilize software not necessarily optimized for real-time performance. For example, running LoCoBot [103] with its pre-packaged `pyrobot` results in planning times exceeding seconds, while *RoWild* delivers millisecond-scale planning on identical hardware (see §5.3.3).

Finally, drawing direct comparisons between *RoWild* and industrial-grade real-time robots is not viable due to the undisclosed specifics of the hardware and software utilized in these robots. However, the scattered data we have been able to gather suggest that *RoWild*'s performance is competitive. It aligns closely with the publicly reported performance numbers of PerceptIn [253], as well as performance numbers measured by other researchers across multiple real-world robots [190].

5.2.5 Measurements

We measure the performance (p) by how inversely proportional it is to the execution time (t); i.e., $p \propto \frac{1}{t}$. After running each program multiple times, we calculate the average execution time based on wall clock time. p quantifies how efficiently a robotic system operates by measuring the speed at which tasks are completed, making it a valuable tool for assessing performance and making direct comparisons between different approaches. In robotics, where real-time responsiveness is often crucial, this metric is particularly essential as it reflects a system's ability to process information and make decisions swiftly.

5.3 The Modeled End-to-End Applications

With the rich set of robotic tasks and algorithms that *RoWild* provides (§5.2), users can model a broad spectrum of robotic applications. In this thesis, we make a concerted effort to model and study a *substantial and representative* range of robotic applications. Detailed in Table 5.4 are the modeled robotic applications, with corresponding references to analogous industrial robots in terms of algorithms and missions, and the tested environments. This is not to say that they utilize identical software. According to data from our industry partners, industrial-grade real-time robots operate on proprietary software infrastructures. These infrastructures, even when bearing core similarities to *RoWild* such as CUDA usage, diverge in facets like library usage. Given the proprietary nature of these systems, specifics are undisclosed and remain out of reach for the broader research community. In contrast, *RoWild* is developed using open-source software, with the goals of maximizing performance, closely emulating real-world robots, and maintaining public availability. The same holds true for the robots' hardware. Many robots do not disclose information about their processors and hardware peripherals for reasons ranging from cyber-attack prevention to competitive edge maintenance.

Table 5.4: The modeled end-to-end applications.

Name	Mission	Resembling	Environment
DeliBot	Delivery	Spot [9]	Our Campus
PatrolBot	Patrolling	Pioneer 3-DX [64]	Our Campus
MoveBot	Manipulation	LoCoBot [103]	Synthetic
HomeBot	Cleaning	Roomba i7+ [79]	Hypersim [218]
FlyBot	Photography	AscTec Pelican [6]	FR Campus [23]
CarriBot	Transportation	Boxbot [10]	Intel Research Lab [40]

Importantly, the robots we model are not only diverse in their applications, ranging from home assistance to campus patrolling, they also demonstrate a variety of computational behaviors. More specifically, we model robots that experience performance bottlenecks at *different stages of the software pipeline*. This offers two significant benefits

to the hardware and systems community:

- It allows the examination of the effects of different techniques across a broader field of robotics. In recent years, several research proposals have targeted one specific robotic pipeline stage (e.g., perception [253], planning [226], and control [220]) and modified the architecture accordingly. While this benefits the specific robotic application studied (e.g., self-driving cars), the impact of such architectural changes has remained unknown for other robotic applications (e.g., home-assistant robots). By providing a benchmark with robots bottlenecked at different pipeline stages, we clarify such impacts.
- Identifying architectural bottlenecks *shared* across robots with differing computational behaviors and devising efficient hardware solutions makes for a more compelling case for adoption of the solutions by industry than addressing bottlenecks arising only in one or two similar robots.

Below, we describe the modeled robots, analyze their computational behaviors, and report the average execution time for *one complete cycle of the software pipeline*, from sensor reading to actuation. This includes the inter-stage communication overheads, though they are typically negligible. Finally, notice while we evaluated the applications in specific environments, they are adaptable to a wide range of other environments.

5.3.1 DeliBot: Legged Robot Delivering Items

DeliBot is a legged robot capable of transporting loads. It navigates CMU's Wean Hall [11] and delivers items to various locations. As the robot already has a map of the building, it does not need to perform any mapping. Figure 5.2.a shows an overview of the application.

Setting

Accurate localization is vital for DeliBot to navigate effectively through the building while avoiding obstacles. The robot uses a *laser rangefinder* and an *odometer* to measure distances to obstacles (blue arrows in Figure 5.2.a) and track its distance traveled (red arrow in Figure 5.2.a), respectively. These sensors work in tandem to provide data for position estimation, allowing the robot to navigate reliably in the building.

DeliBot uses Monte Carlo Localization (MCL) [254], a.k.a. particle filter, to accurately determine its location and orientation within the building. As explained in §4.2.1, MCL is a robust technique that can handle various sensor models (e.g., non-linear, non-Gaussian), which makes it suitable for complex environments like the building's narrow corridors.

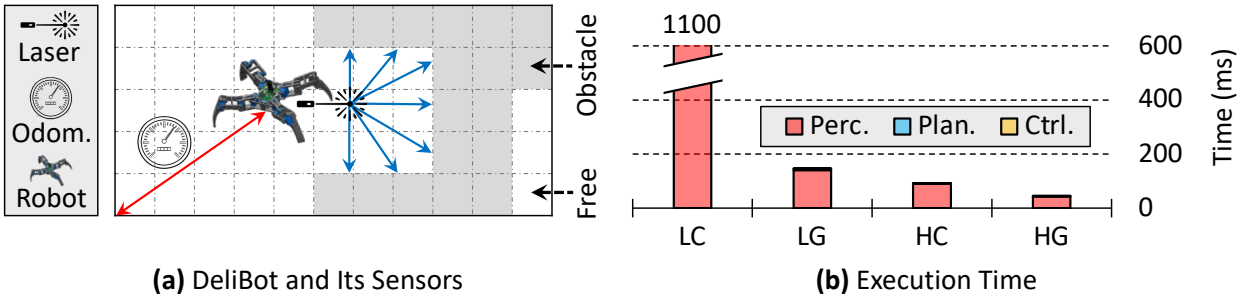


Figure 5.2: DeliBot operating in CMU's Wean Hall.

We fine-tune MCL's parameters (e.g., the number of particles) such that it achieves a localization error—the difference between the robot's estimated and actual location—of a few centimeters or less, which is satisfactory for the application.

Because the environment is known and structured, DeliBot's planning is relatively simple, especially as the application does not require an optimal path. The robot simply selects a set of waypoints and navigates between them using a greedy search algorithm that directs it towards the nearest one.

Similarly, DeliBot's control stage is straightforward, with the controller simply selecting the leg to pull the robot forward. DeliBot's limited dynamics, such as fixed velocity and acceleration, contribute to the ease of control.

Evaluation

Figure 5.2.b shows DeliBot's compute time on different platforms, where the planning and control stages are processed only on CPUs, while the perception stage leverages GPUs when available. The perception stage is the main contributor to the end-to-end latency, taking more than 97% of execution time. To ensure high localization accuracy in complex environments like the evaluated building, maintaining numerous particles in MCL is required. This improves accuracy by offsetting the effect of noisy measurements during the resampling process (§5.3.1), but it also results in a higher computational workload, making perception the computational bottleneck.

Fortunately, particles are independent and can be processed in parallel, which GPUs exploit to offer significantly higher performance compared to CPUs. Additionally, as particles converge and their hypotheses align, their control flow in response to sensor readings becomes similar, reducing branch divergence to near-zero, helping GPUs to offer superior performance during the steady stage. Comparing end-to-end times, *LG*/*HG* outperforms *LC*/*HC* by 7.9×/2.1×.

Similar to observations made in §4.2.1, *ray-casting* is the time-dominant operation of MCL, accounting for 58%–83% of perception latency. It is performed in order to correspond laser measurements with particles' hypotheses and involves a cell-by-cell traversal of the environment map in different directions.

Perhaps counter-intuitively, we find CPU vectorization ineffective for ray-casting. The reason being the generation

of axes-unaligned memory accesses (blue arrows in Figure 5.2.a) due to the *varying orientation* of the robot with respect to the environment axes, which current vectorization engines cannot handle. Further details on this observation are provided in §5.4.1.

Finally, as DeliBot employs a simple planner and controller, their impact on the end-to-end execution time is insignificant (up to 2.3% of execution time).

On the memory front, when the application is effectively parallelized, the memory bandwidth consumption can become substantial. On the low-end platforms, this reaches a peak bandwidth utilization of 4GB/s, which is significant (see §5.4.2). On the high-end platforms, while the absolute memory bandwidth consumption is even greater due to increased parallelism, the relative impact on performance is less pronounced due to the expansive bandwidth resources inherent to the hardware.

Takeaway

• DeliBot's perception stage is a bottleneck, as it requires costly ray-casting operations within hundreds of particles to achieve acceptable localization accuracy in challenging environments, such as the narrow corridors of our building.

• LC's limited support for parallelism results in poor performance, making it unsuitable for time-critical applications with high accuracy demands.

5.3.2 PatrolBot: Wheeled Robot Patrolling Campus

PatrolBot is a wheeled robot that is responsible for patrolling our campus and detecting suspicious objects. Due to the dynamic nature of the environment, PatrolBot does not rely on a static map to navigate. Instead, it performs simultaneous localization and mapping (SLAM), which involves creating a map of the environment as it moves around and determining its own position in that map. To accomplish this task, PatrolBot uses six identified landmarks throughout the campus. An overview of the modeled application is shown in Figure 5.3.a.

Setting

PatrolBot is equipped with *range* and *angle* sensors, which provide distance and angle measurements relative to the landmarks, respectively. PatrolBot uses these measurements as input to the Extended Kalman Filter (EKF) algorithm [240] to solve the SLAM problem.

As explained in §4.2.2, EKF is a popular SLAM algorithm handling various sensors and providing accurate estimates of the robot's position and the environment's map, despite noise and uncertainties. It linearizes non-linear motion and measurement models, enabling a Gaussian probability distribution to represent the system's state. This allows recursive state estimation using the Bayes' rule, which is computationally efficient and capable of handling uncertainties in the measurements.

While performing SLAM, PatrolBot also analyzes images from the surrounding area to detect suspicious objects. To accomplish this, it uses MobileNet-SSD [252], a lightweight neural network (NN) feature extraction algorithm, pre-trained on the MS COCO dataset [185]. MobileNet-SSD predicts object bounding boxes and classes using classifiers, and it is optimized for mobile devices with high accuracy. MS COCO is a widely-used, large-scale dataset with over 80 object categories, commonly used for image recognition.

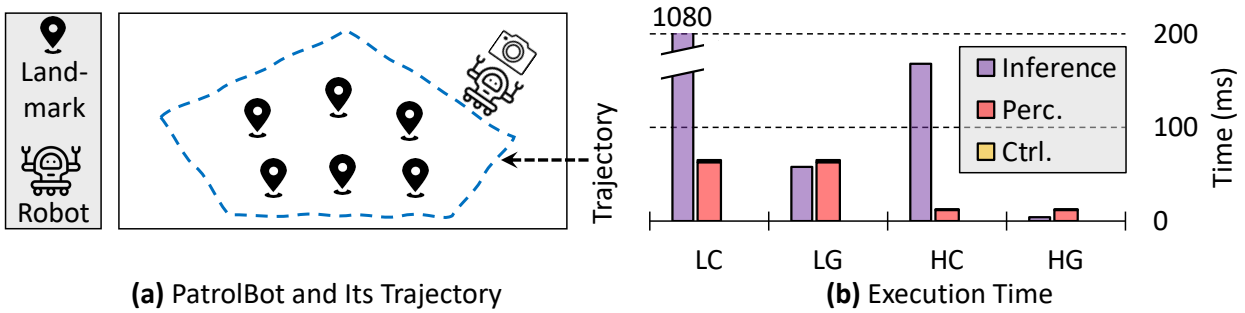


Figure 5.3: PatrolBot operating on CMU campus.

If the NN cannot confidently categorize an object close to the robot or if it detects weaponry, PatrolBot infers that the object may be suspicious. If the object remains in the robot's view for a certain time, PatrolBot alerts the security personnel.

PatrolBot follows a pre-determined path around the landmarks, without the need for online path planning. It utilizes the Pure Pursuit algorithm [209] to determine the optimal steering angle and speed for efficiently following the pre-determined path. Also, PatrolBot is equipped to detect obstacles, and it automatically stops and resumes its movements accordingly to avoid collisions.

Evaluation

Figure 5.3.b shows PatrolBot' compute time across platforms. Notice, NN inference runs concurrently with the robot's software pipeline, not as a part of it. Therefore, the robot's overall performance is determined by the greater of the pipeline latency or the NN inference latency. The software pipeline runs entirely on CPUs, while the NN inference runs on GPUs when available.

On CPUs, NN inference determines the robot's performance. On GPUs, however, the inference is significantly accelerated, leading to the performance bottleneck being shifted to robot perception. GPUs run NN inference 6.5×–336× faster than CPUs, due to their massively-parallel architectures.

The perception stage, which runs EKF, is the second major compute component of PatrolBot. As discussed in §4.2.2, EKF heavily utilizes linear algebra, for instance, in computing the Jacobian matrix of state variables and propagating the covariance matrix to depict state uncertainty. The size of matrices and accordingly EKF's execution

time scale with increasing the number of measurements; with six landmarks and two measurements per landmark in our setup, EKF becomes moderately time-consuming.

Finally, Pure Pursuit is relatively simple and fast, taking up to 3.2% of the end-to-end execution time. It calculates the robot's steering angle, which only requires computationally simple geometric calculations such as distance and angle calculations.

Takeaway

• Multiple compute platforms greatly improve a robot's real-time performance. LG, priced at $149, outperforms HC, priced at $1493, by running the NN on GPU and the rest on CPU. However, without the GPU (i.e., LC), it lags significantly.

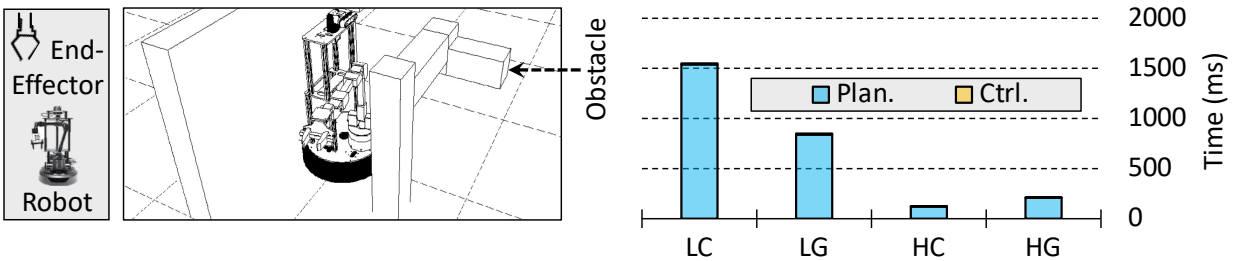
5.3.3 MoveBot: Arm Manipulator Moving Items

MoveBot is a manipulator equipped with a 5-degree-of-freedom (5-DoF) robotic arm that excels at moving objects with speed and precision. Its real-world applications are primarily in the manufacturing and warehousing industries, where products need to be moved quickly and efficiently. We evaluate MoveBot in a cluttered, synthetic environment, depicted in Figure 5.4.a. We carefully model MoveBot after our in-house, open-source LoCoBot [103].

Setting

The environment is mostly static: the robot's base and the obstacles (walls) remain stationary. However, items are relocated by the robot itself. Hence, the robot performs perception only once offline, and keeps track of the moved objects to avoid collisions at runtime.

MoveBot is repeatedly queried to pick and place items. Upon a query, it performs path planning to move its end-effector to the designated position while avoiding obstacles. The planning is done in the joint angle space of the arm, which has five links. The planner finds a sequence of angles $(\alpha_1, \alpha_2, \ldots, \alpha_5)$ that moves the end-effector towards the goal position. Since the configuration space is high-dimensional, MoveBot's planner *samples* the space to find a path in a reasonable time.



(a) MoveBot and The Modeled Obstacles **(b)** Execution Time

Figure 5.4: MoveBot operating in a synthetic environment.

As explained in §4.2.8, Rapidly-exploring Random Trees (RRT) [179] is a popular algorithm for high-dimensional planning. Chung et al. [137] fabricate a full-fledged RRT-based path planning processor in 40nm CMOS. RRT builds a tree from the start point to the goal by randomly sampling the configuration space and connecting the samples to the nearest nodes in the tree, ensuring that the path is collision-free. This requires numerous *collision detection* operations to check whether the robot will collide with obstacles as it moves from one configuration to another.

Accurate collision detection is a time-consuming process, as discussed in §4.2.4 and relevant studies [123, 160, 226]. While hardware accelerators have been proposed [160, 182, 200, 226], they are not widely adopted yet. On the software side, "reworked" methods have been developed that bound obstacles in the environment with simpler shapes like cubes and exploit their geometric properties to speed up collision detection [142]. While these methods underestimate free space, they offer far faster performance. For MoveBot, we use the *cuboid-cuboid collision detection (CCCD)* algorithm [142], which bounds obstacles and the robot body with cubes and checks for intersection during movement planning. We find that CCCD strikes a good balance between accuracy and execution time for MoveBot.

Finally, MoveBot employs a simple PID controller [247] to move its arm by taking desired joint positions as input and generating control signals to the motors based on the difference between desired and actual joint positions.

Evaluation

Figure 5.4.b shows MoveBot's compute time on different platforms, where the control stage runs on CPUs, while the planning stage leverages GPUs when available.

Collision detection, even in its reworked form, remains the most time-consuming aspect of planning, taking more than 85% of the end-to-end time. Each collision detection involves checking the intersection of four cubes around the robot's body (joints and peripherals) with nine cubes surrounding obstacles. *LC* offers limited parallelism support, resulting in the lowest performance, while *HC* utilizes both coarse-grained and fine-grained parallelism to achieve high performance. Coarse-grained parallelism occurs when different cuboid-cuboid checks are performed in parallel using different threads. Fine-grained parallelism happens because *HC*'s large instruction window can extract high instruction-level parallelism (ILP) from operations like finding multiple *independent* neighbors at each RRT iteration (see §4.2.8).

We use Bialkowski et al.'s approach [123] to parallelize collision detections on GPUs. This improves the performance of collision detection, especially on *HG*, which has larger on-chip caches and more cores. However, significant branch divergence occurs during collision detection. Our profiling with NVIDIA Nsight Compute [58] reveals that over 18% of branches are divergent, which is considered high [169]. This is due to CCCD conducting *multiple* checks (e.g., edges, normals, axes) on every configuration to determine collision status, and the outcome of checks differ, resulting in taking divergent flows. As the configurations are dynamically shaped at runtime, warp-synchronous programming is not applicable, and further improvements require dynamic techniques.

Finally, PID controller is simple and fast, taking up to 1.6% of execution time. It uses a linear combination of three terms, proportional, integral, and derivative, to calculate the control signal. Each term needs only basic arithmetic operations, such as subtraction, making the algorithm computationally cheap.

Robotic arm manipulators' memory demands can become substantial when they requires a large number of drawn samples. This often arises when the planner necessitates a *fine resolution* to ensure safe and accurate navigation in densely packed settings. Additionally, as DoF increase, memory consumption grows exponentially—a phenomenon termed the "curse of dimensionality." Even a minor uptick in DoF can drastically amplify the potential configurations the planner must account for, thereby escalating memory demands.

However, for the given application model and evaluation setting, the memory system is not the primary constraint. In the evaluated platforms, the 5-DoF robot operating within the tested environment does not significantly tax the memory bandwidth, with the exception of *LG*. On *LG*, memory is communally allocated between the host and the device. When collision detection undergoes massive parallelization on the GPU, the surge in memory requests places substantial strain on the bandwidth. This results in a notable performance drop, as elaborated in §5.4.2.

Takeaway

• *Collision detection is costly and can pose a risk to real-time constraints, especially when running on low-end platforms. Safety-critical applications use a separate path to throttle robots directly in case of real-time collision detection failure [253].*

5.3.4 HomeBot: Assistant Robot Cleaning House

HomeBot is a home-assistant robot performing missions like collecting debris. We use `ai_001_001` from Hypersim [218] (shown in Figure 5.5.a) to model a dense indoor environment. HomeBot mimics the functionality of home-cleaning robots like Roomba [79].

Setting

The environment is dense and dynamic. To have an accurate and up-to-date understanding of the environment, HomeBot captures a series of images by its RGB-D camera, as it moves. These images are then processed to create a 3D model of the scene (*3D scene reconstruction (3DSR)*). HomeBot employs point-based fusion [167] to perform 3DSR. As explained in §4.2.3, this process involves using point clouds, collections of 3D points that represent surfaces and features in a scene, and merging them together to create a detailed map of the environment. To achieve this, the method needs to *align the point clouds*: it finds the best transformation between two overlapping point clouds and then iteratively refines the alignment until the two clouds are as closely aligned as possible. After that, the point clouds are fused into a single map; HomeBot uses this map to perform its missions.

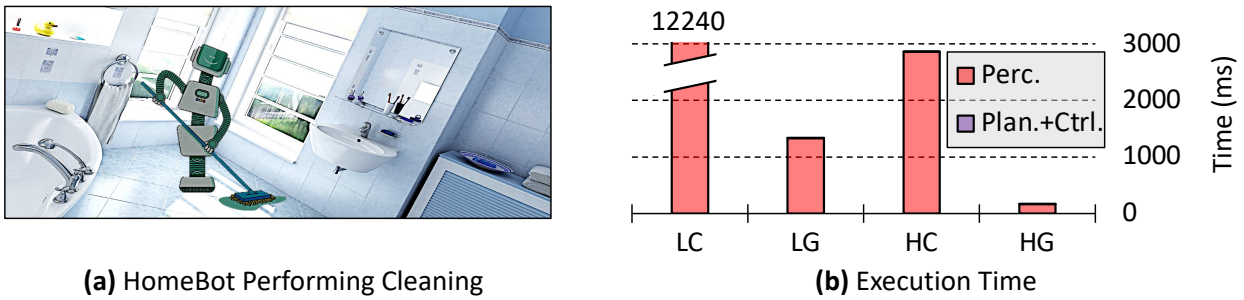


Figure 5.5: HomeBot operating in a benchmark environment [218].

During the alignment of point clouds, nearest-neighbor search (NNS) is performed to find the closest points in one point cloud to each point in the other point cloud. The goal of this search is to identify the best possible correspondences between points, which are pairs of points that are likely to represent the same physical feature in the real world.

The images are 768×1024 pixels, high-quality enough for accurate mapping. We run our experiments on 100 frames of the dataset.

HomeBot employs a reactive planning and control to clean the house, where the cleaning method is determined based on the size of the dirty spot. For small stains, the robot applies a small amount of cleaning solution and uses brushes, while for larger spots, it switches to vacuuming. Once the spot is clean, the robot moves along the free paths to check the rest of the house.

To implement the reactive approach, we use Behavior Tree (BT) [154]. BT is a computational model that is widely used for planning and control in robotics. It is structured as a tree consisting of interconnected nodes, which define conditions (e.g., small or large stain) and actions (e.g., brushing or vacuuming) that must be taken in response to external stimuli.

Evaluation

Figure 5.5.b shows HomeBot's compute time on different platforms, where planning and control run on CPUs, and perception runs on GPUs when available. Perception, which performs 3DSR, takes more than 99% of the entire execution time. 3DSR is a hugely data-intensive and highly demanding task. With higher-resolution images, e.g., 12-megapixel iPhone 14 images, 3DSR will be even more intensive.

As discussed in §4.2.3, the irregular nature of the algorithm and data structures involved in 3DSR generate *irregular data references*, rendering it a memory-bound task. During reconstruction, the algorithm needs to process the 3D point cloud data, leading to accessing data at irregular intervals based on the point locations being processed. Similarly, in the alignment stage (§5.3.4), the algorithm needs to match and align multiple point clouds from various views, which requires accessing data in non-linear ways—3D points that are *semantically close* to each other in the

scene, can be spread *arbitrarily* across the memory layout.

GPUs' high parallelization power make them a superior platform for 3DSR. By partitioning the data and processing each partition simultaneously, 3DSR's operations such as alignment are parallelized, resulting in a significant acceleration—GPUs outperform CPUs by up to 17×.

Moreover, *HG* outperforms *LG* by a factor of 8 due to its larger memory capacity and bandwidth, which can handle the bandwidth-intensive irregular data references generated during 3DSR. On the other hand, *LG* has a weaker memory system that hits the "memory wall" when fully parallelizing the task—the concurrent allocation of large thread-private dynamic data objects and their irregular referencing creates an overwhelming demand for the limited memory capacity and bandwidth of *LG*. §5.4.2 provides details on this phenomenon.

Finally, BT is fast, taking up to 0.6% of the end-to-end execution, since it selects parameters from a *discrete* set of options instead of a continuous spectrum. This eliminates the need for sophisticated methods to determine optimal parameters, making the algorithm computationally cheap.

Takeaway

• 3DSR is a resource-intensive task that can only be harnessed by HG to achieve real-time performance while being accurate.

• The performance of the memory system is crucial for HomeBot, particularly in its perception stage, which involves many irregular data accesses.

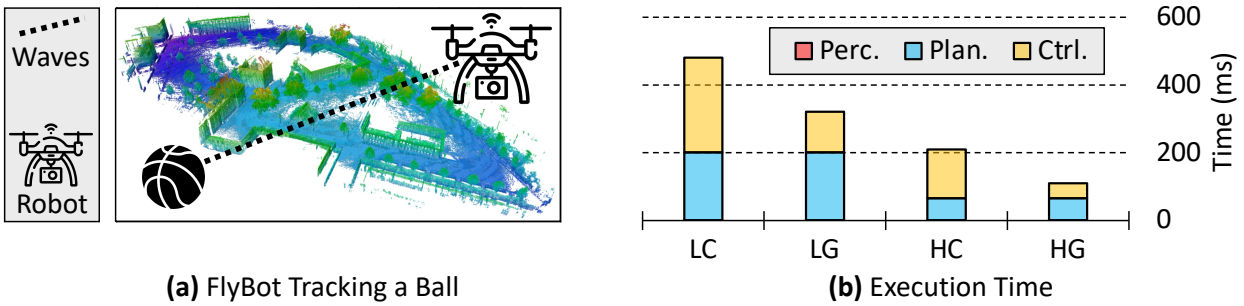
5.3.5 FlyBot: Drone Performing Aerial Photography

FlyBot is a pilotless drone that covers sports events on the Freiburg campus [23], tracking a specific object such as a ball and adjusting its position to maintain a seamless view of the target area. Figure 5.6.a shows an overview of the environment.

Setting

FlyBot uses a radar sensor to track the ball's position and maintain it in the camera frame in real-time. The sensor emits radio waves that bounce back from the ball to determine its location, based on its radar cross-section, which measures its reflectivity to radar waves. The sensor is calibrated to detect only the ball, as it bounces off the ball and not on other objects, which absorb or scatter the waves.

A lookup table approach, similar to [199], is employed to determine the position of the ball quickly based on sensor data. The table is created by storing readings from the radar sensor for different ball positions offline. During online detection, the table is used to locate the ball. While the table's accuracy may not be high, its performance justifies its use in this application where the exact ball location is not crucial.



(a) FlyBot Tracking a Ball **(b)** Execution Time

Figure 5.6: FlyBot operating in a benchmark environment [23].

FlyBot uses the weighted A⋆ (WA⋆) [216] algorithm to plan paths in the (x,y,z) space. As explained in §4.2.6, WA⋆ uses a heuristic function to guide the search towards the goal. It creates an environment graph where each node is assigned a cost representing its distance from the starting point. A weight factor, ε, is applied to the heuristic function to balance the exploration and exploitation of the search space: large ε values prioritize faster search, while small ε values prioritize a more optimal path. We use Euclidean norm as the heuristic function and set $\varepsilon = 4$.

FlyBot utilizes model predictive control (MPC) [144] to follow the path identified by the planner. As explained in §4.2.14, MPC formulates motion control as a *convex optimization problem*, solving it to identify a feasible path that adheres to the robot's constraints. By controlling FlyBot's velocity and acceleration, MPC ensures the robot stays on course while minimizing any deviations from the planned path.

Evaluation

Figure 5.6.b shows FlyBot's compute time on different platforms, where the perception and planning stages run on CPUs, while the control stage runs on GPUs when available. Both planning and control stages contribute significantly to the overall pipeline latency. Planning takes 31%–63%, and control takes 37%–68% of the execution time, depending on the platform. The perception stage involves a computationally cheap table lookup, taking less than 1% of the end-to-end time.

During path planning, the environment graph is searched by generating nodes corresponding to (x,y,z) locations within the environment area and checking whether they are free from collisions. The checking of nodes for collision is inexpensive due to the altitude at which FlyBot operates, which is relatively free from clutter. Therefore, the time taken for path planning is largely determined by the generation of nodes, which is proportional to the size of the environment.

WA⋆ is difficult to parallelize, especially when collision detection is inexpensive (see §4.2.5). The overheads of multi-threading (e.g., locks) may not be justified by the relatively light workloads of nodes. Consequently, single-thread performance is crucial, and this is why high-end computing platforms far outperform low-end ones—the instruction-per-clock (IPC) of *HC* is 2.8, while that of *LC* is 1.2.

The MPC algorithm for generating control commands is expensive. RoBoX [220] proposes a full-fledged hardware for accelerating this process. More than 80% of the control time is spent on solving a constrained vector-valued function through an optimization process. This process involves solving a set of linear equations at each time step, but it can be parallelized and solved through smaller sub-problems. GPUs can take advantage of this parallelism, resulting in high performance.

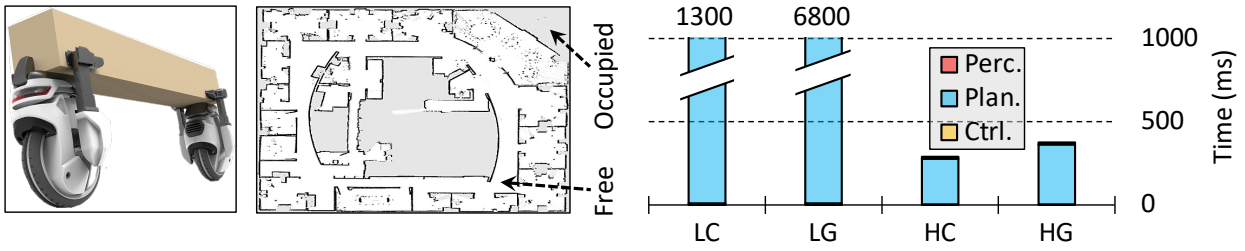
Notably, MPC benefits from CPU vectorization due to the optimization process involving solving linear equations and performing matrix factorization—both can be readily vectorized. The CPU can perform multiple calculations at once by utilizing SIMD instructions, such as the AVX-512 instructions available in *HC*, resulting in faster computation (see §5.4.1).

Takeaway

- *FlyBot's performance is bottlenecked by both planning and control stages, with throughput determined by the longest stage. This is influenced by the presence or absence of GPUs in the planning and control platforms.*
- *The planning stage of FlyBot requires high single-threaded performance.*

5.3.6 CarriBot: Driverless Vehicle Transporting Goods

CarriBot is a driverless vehicle designed to optimize transport processes for time-sensitive materials such as those used in chip manufacturing. In our modeled application, CarriBot navigates Intel Research Lab [40] and carries goods, minimizing travel time while avoiding obstacles. Figure 5.7.a shows an overview of the application.



(a) CarriBot and The Environment Map **(b)** Execution Time

Figure 5.7: CarriBot operating in a benchmark environment [40].

Setting

CarriBot operates in a known area: the factory's aerial map is furnished to the robot ahead of transportation. Nevertheless, it must perform real-time mapping to comprehend its immediate vicinity, including the workers and other vehicles in the area, thereby avoiding collisions.

A monocular camera is mounted on CarriBot to perform mapping. The camera captures images of the environment, which are then processed to generate "occupancy grids."

Occupancy grids are a way of representing the environment as a grid of cells, where each cell is assigned a value indicating whether it is occupied by an obstacle or not. We use Probabilistic Occupancy Map (POM) [151] to generate occupancy grids from camera images. POM divides the environment into a grid of cells and assigns each cell a probability value that represents the likelihood of it being occupied by an obstacle; cells with probability values above a certain threshold are deemed to be occupied, while those below it are considered to be free.

CarriBot utilizes an Inertial Measurement Unit (IMU) [192] consisting of accelerometers, gyroscopes, and magnetometers. The accelerometers measure linear acceleration, the gyroscopes measure the rate of rotation around axes, and the magnetometers measure the Earth's magnetic field, aiding in determining the vehicle's orientation relative to the magnetic north pole. The EKF algorithm (see §5.3.2) processes the IMU data to localize the vehicle (without mapping).

CarriBot explores the (x, y, θ) space to find the shortest path to the goal. Due to the presence of narrow pathways in the environment and the use of a probabilistic map, CarriBot uses accurate brute-force collision detection to avoid any collision. Different algorithms are used for different platforms: WA* (see §5.3.5) is used for low-end devices, while RA* [118] and GA* [256] are used for *HC* and *HG*, respectively. All the algorithms use the Euclidean distance heuristic with $\varepsilon = 1$.

RA* and GA* improve parallelism in the search process by introducing speculative parallelism on CPUs and proposing a new parallelization method on GPUs, respectively. They improve performance when collision detection is costly (unlike §5.3.5), but are resource-intensive and unsuitable for low-end devices. Notice, they are research state-of-the-art, not established algorithms. We include them to showcase the potential of powerful computing systems to enable sophisticated algorithms for significant speedups in future robotics. RA* is a contribution of this thesis and will be thoroughly explained in §6.6.

CarriBot uses Dynamic Movement Primitive (DMP) [175] to control its movements along the planned path. DMP is a machine learning technique that decomposes a movement into simpler sub-movements. It uses Gaussian bias functions and shape parameters to *define* the trajectory shape. These parameters are acquired through *imitation learning* [161] and linear regression. Once the parameters are acquired, the final trajectory, including velocity and acceleration, is computed.

Evaluation

Figure 5.7.b shows CarriBot's compute time across the platforms, where perception and control run on CPUs, and planning runs on GPUs when available. As noted above, *HC* and *HG* run different planning algorithms.

Processing monocular camera images using POM is computationally cheap, as it relies on a simple probabilistic model instead of expensive 3D reconstruction (see §5.3.4). Similarly, EKF-based localization is also simple. Unlike EKF-based SLAM (see §5.3.2), EKF-based localization only estimates the robot's state without including all the

landmarks on the map. This significantly reduces the state space (3 variables instead of 15), resulting in a large reduction in compute workload. Overall, perception accounts for less than 1% of the end-to-end execution time.

Path planning is the major component of computation time across all platforms, with collision detection taking up more than 80% of the time. As explained in §4.2.4, this process involves checking many environment locations corresponding to the projected vehicle location to ensure no collision occurs during planning, which is resource-intensive due to the relatively large robot body in the operating environment.

RA* and GA* significantly improve search performance by parallelizing intensive collision detection operations on high-end processors, but their resource demands render them unsuitable for low-end devices. RA* lacks the requisite number of cores for speculative parallelization on *LC*, while *LG fails* to run GA* due to its limited memory capacity; the algorithm needs to maintain thousands of min heaps to find the solution in parallel, which overwhelms the existing memory capacity of *LG* (see §5.4.2). RA* on *HC* outperforms WA* on *LC* by 4.6×, and GA* on *HG* outperforms WA* on *LG* by 18.2×.

We find CPU vectorization ineffective for collision detection, despite the memory locations checked being nearby and running the same check. This is due to the arbitrary orientations of the robot during planning rendering the layout of memory references axes-unaligned (similar to ray-casting; see §5.3.1), which makes vectorization inapplicable (see §5.4.1 for details).

Also, the search algorithm's use of irregular data structures like min heaps causes many irregular cache misses, beyond the realm of the evaluated processors' prefetchers (see §5.4.3). Cache misses are significantly more in GA*, as it employs *thousands* of min heaps, irregularly traversing them.

Finally, DMP's online computation is inexpensive due to its offline learning of motor control primitives, which are stored as weights in a set of basis functions. Online execution involves simply generating new movements using a weighted sum of these basis functions, which is computationally inexpensive: up to 4.1% of the end-to-end execution time.

Takeaway

• *Accurate collision detection in planning can be computationally demanding, and low-end platforms may not offer sufficient performance for time-critical applications.*

• *Algorithmic advancements can lead to the full utilization of hardware potential, resulting in transformative improvements in robotics.*

5.4 Architectural Implications of Robotics

Robotics involves diverse workloads, and no single hardware can serve all of them equally well. It is unlikely that vendors will fabricate a specific processor for each robotic application. However, designing processors based on *shared*

characteristics across a range of robotic workloads is practical. This section studies the system- and hardware-level implications of robotics, gaining insights into designing "robotics processors."

5.4.1 Vectorization and Irregular Memory Layouts

All modern CPUs feature vector instructions and large vector registers, such as *LC* and *HC* with 2KB and 44KB vector registers respectively. Vectorization is shown to remarkably enhance performance across various workloads.

Figure 5.8 shows execution time using manual and compiler vectorization, normalized to that without vectorization. We write vectorized code using VCL [108], utilizing input from Intel Advisor [31]. Compiler vectorization is primarily used to enhance the vectorization of third-party codes.

We conclude that, despite the large silicon it occupies, *vectorization does little for most robotic workloads*. FlyBot is an exception to this trend, experiencing 5% speedup with vectorization. FlyBot benefits from vectorization, as it optimizes a vector-valued function in its control stage, as explained in §5.3.5.

Our investigation shows that vectorization is ineffective for operations like ray-casting and collision detection, despite their perceived suitability. The *varying orientation* of the robot with respect to the environment axes leads to axes-unaligned memory accesses (blue arrows in Figure 5.2.a), which current vectorization engines cannot handle. Notice, the `VPMULTISHIFTQB` instruction, introduced in Intel's Cannon Lake, still cannot capture these patterns as the generated memory references exceed the quadword range. This issue will become more significant for future advanced robots, such as surgical robots that require higher accuracy: operations like collision detection will be more computationally intensive, but current vectorization approaches will not provide a solution.

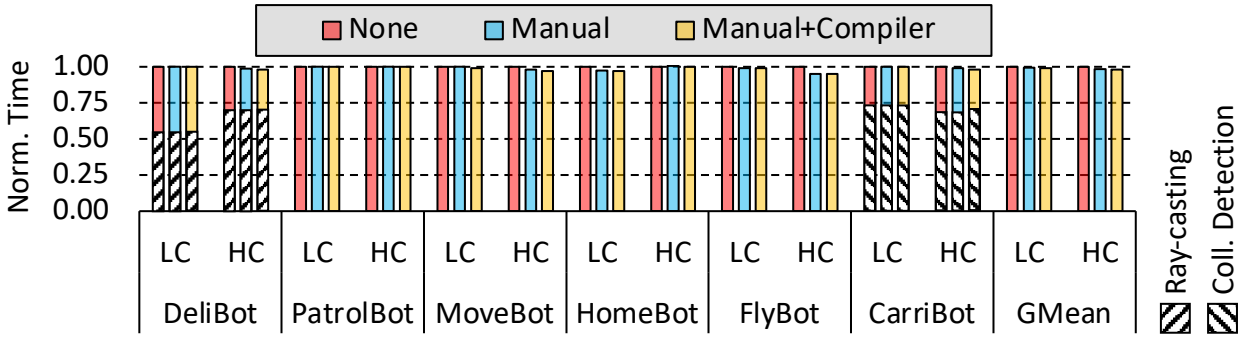


Figure 5.8: Vectorization's impact on *LC* and *HC*.

To address this issue, current vector architectures can incorporate an *additional operand* into vector instructions: a register containing the traversal orientation. This would enable the hardware to compute the memory addresses corresponding to the robot's current layout and vectorize fetching them. In §7.4.2, we evaluate this approach and shed light on its effectiveness.

5.4.2 Parallelism and The Memory Barrier

We find that massive parallelism on the edge, i.e., *LG*, hits the memory wall. The memory problem is twofold: *(i)* limited DRAM capacity and *(ii)* limited DRAM bandwidth.

In detail, we observe that massively parallelizing memory-intensive applications like HomeBot overwhelms the limited DRAM capacity: in-parallel allocation of mega-scale thread-private data (e.g., bookkeeping 3D pixels) dwarfs the available few gigabytes of DRAM. Further, the limited DRAM capacity prevents employing some massively-parallel algorithms (see §5.3.6).

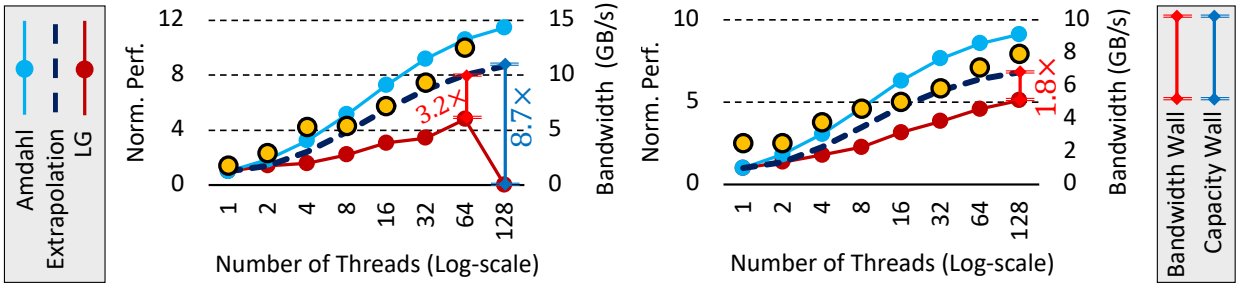


Figure 5.9: Performance scaling hits the memory wall. The difference between Amdahl and *LG* is named AG, and the difference between Amdahl and Extrapolation is named EAG.

Additionally, on-edge parallelism encounters the memory bandwidth wall, where performance scaling is limited even with increasing cores. Figure 5.9 shows the performance of HomeBot, *RoWild*'s most memory-intensive application (on the left side of the graph), and the average of three other memory-intensive applications, namely DeliBot, MoveBot, and FlyBot (on the right side of the graph), with varying the number of threads normalized to single-thread performance. The corresponding bandwidth utilization is shown by yellow dots on the graph.

The complex architecture of GPUs makes it challenging to precisely quantify the impact of memory bandwidth on performance scaling. To estimate this impact, we employ the following methodology: First, we subtract the execution time of the program's serial components from the total execution time to isolate the execution time of the parallel section (t_p). Then, we compute the disparity between the execution time of an n-threaded execution and ideal scaling as defined by Amdahl's law ($\frac{t_p}{n}$); this disparity is termed the *architecture gap (AG)*. Using a linear function, we extrapolate AG from when the bandwidth utilization is low (involving just a few threads) to when it is high (with all threads in use); we name it the *extrapolated AG (EAG)*. Because EAG is derived from scenarios featuring low bandwidth, it is likely *unaffected* by the memory bandwidth wall. Consequently, the contrast between EAG and AG in high-bandwidth settings effectively represents the impact of the memory bandwidth wall.

The results show that *performance scaling on the edge is hindered by the memory wall*. The memory capacity/bandwidth wall alone prevents up to 8.7×/3.2× performance scaling (in HomeBot).

It is worth noting that certain edge platforms have recognized the need for increased memory bandwidth and

have substantially improved their offerings. For example, NVIDIA's Jetson AGX Orin [45] provides a remarkable 204.8GB/s of memory bandwidth, which is eight times more than *LG*'s offering. However, this enhancement comes at a considerable cost difference, with AGX Orin's solution priced at $1999 compared to *LG*'s $149, making it cost-prohibitive for many robotic applications. Therefore, we advocate for system-level techniques, such as enhanced data sharing, to improve bandwidth utilization and achieve cost-effective performance scaling. Lastly, high-end platforms like *HC* and *HG* are already overprovisioned in terms of memory bandwidth and do not exhibit this issue.

5.4.3 Data Prefetching and Complex Access Patterns

LC and *HC* include *simple* stride/stream hardware prefetchers. Also, software prefetching is implemented through compiler optimizations and manual profiler-driven programming (see §5.2.2). Figure 5.10 shows the execution time with prefetching normalized to without it. Notice, hit ratio is not a good indicator of prefetching effectiveness in *HC*, as different prefetchers are for *different* cache levels: `Hardware Prefetcher` (L2), `Adjacent Cache Prefetch` (L1), `DCU Streamer Prefetcher` (L1), `DCU IP Prefetcher` (L1), and `LLC Prefetcher` (L3).

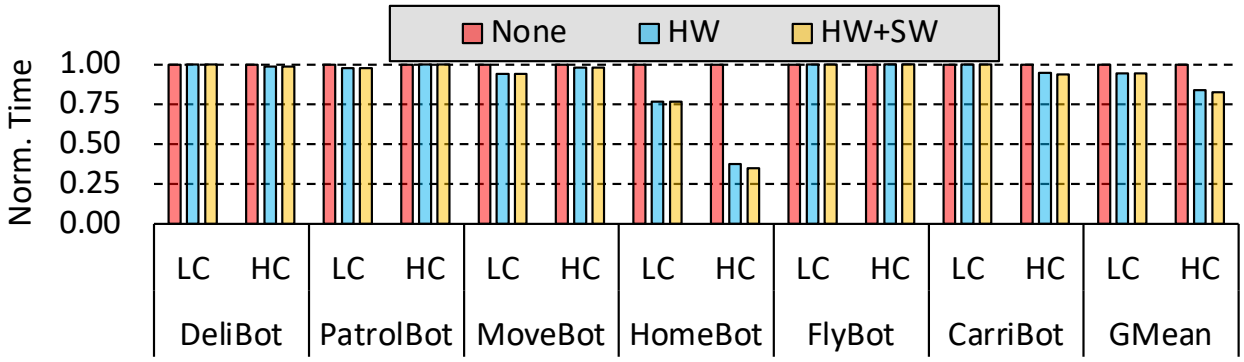


Figure 5.10: Prefetching's impact on *LC* and *HC*.

The results show disabling hardware prefetchers causes a significant slowdown: *stride prefetching is effective for robotics*. This contrasts with prior works which find stride prefetching to be highly ineffective for various workloads, including scientific, commercial, and cloud [147].

Stride prefetching is effective for two reasons: first, many robotic computations, such as linear algebra, inherently involve strided memory accesses. Second, we deliberately use static data structures, such as arrays, instead of dynamic ones like linked-lists, wherever possible to enable memory accesses to occur in fixed-sized blocks due to the contiguous nature of memory locations. While this approach may reduce the generality of each program's binary and require recompilation for different robots, it is justified by the performance improvement and the application nature, where the robot is aware of its components (e.g., camera) and can specialize software based on them.

Nevertheless, *simple prefetchers are inadequate for memory-intensive robotic applications*, as they do not capture complex memory patterns generated by the employed irregular data structures like min heaps and hashmaps. This

results in a large post-prefetch LLC misses per kilo instruction (MPKI), with CarriBot having an MPKI of 1.5 (2.1) and HomeBot having an MPKI of 3.5 (1.4) on *LC* (*HC*, respectively).

Finally, software prefetching offers little improvements. In DeliBot, PatrolBot, FlyBot, and CarriBot, there is insignificant opportunity for software prefetching. In MoveBot and HomeBot, there is some opportunity, but prefetches are not timely–they hide the latency of one or a few loop iterations. In HomeBot, additionally, excessive software prefetching can degrade performance due to a full L1D fill buffer of pending misses.

5.4.4 Caches and Data Movements

Prior work [187] reports "data movement" as the primary cause of slowdown and inefficiency in processors. Figure 5.11 shows the percentage of used bytes out of all fetched bytes for every program. We get the trace of memory accesses using Intel Pin 3.25 [63] and process them offline. We consider 64B cachelines, which is the configuration of the CPUs.

The results show that *caches perform excessive unnecessary data movements (UDMs)*. In HomeBot, only 33% of the fetched data is used; and, on average across all, only 56% of the fetched data is used (cf. 76%, 92%, 80% average utilization reported for SQL [135], scientific [134], SPEC [168] workloads, respectively). This shows how caches are overprovisioned and are unaware of robotic semantics.

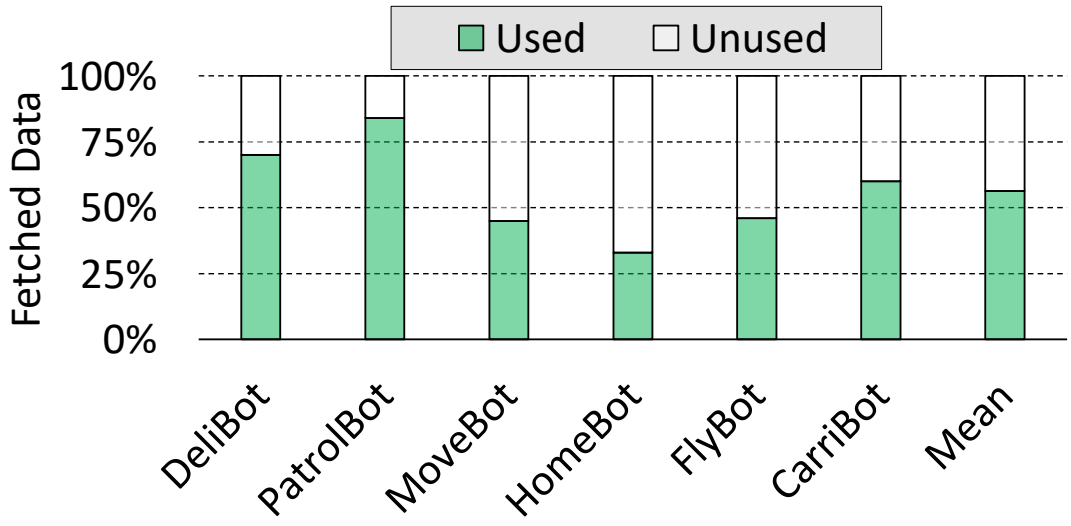


Figure 5.11: Data movements breakdown.

Caches are architected partly based on the "traditional spatial locality" concept; they use large blocks to capture *(i)* large data types (e.g., 8-byte `double`), and *(ii)* nearby data items (e.g., 1D arrays). However, memory accesses are not always so in robotics. For example, in ray-casting and collision detection—two dominant kernels—memory is traversed mostly in an *oriented* manner (see §5.4.1), and every memory request accesses only *one bit*; caches are in fact working *against* application semantics, resulting in significant UDMs.

Notice, in this experiment, we do not model *(i)* hardware prefetches (Pin traces lack) and *(ii)* cache size (infinite-size cache is modeled). Hence, 44% useless fetches we report is an *underestimate*. Also, with larger cachelines (e.g., in GPUs), UDMs increase; e.g., by 1.22× with 128B lines.

To address this, caches need awareness of the robot's body and traversal. This could be achieved by fetching likely-to-be-used parts of cachelines or dynamically adjusting cache block sizes. Either way, the semantics must be transferred to hardware.

5.5 Chapter Summary

In this chapter, we introduced *RoWild*, a comprehensive, open-source benchmark suite for robotics, and used it to conduct a systematic study of robotics on modern hardware. With this suite and this study, we aim to drive research at the intersection of robotics and systems to improve the performance of robotics. The open-source nature of *RoWild* facilitates widespread access and adoption among researchers and practitioners in the computer architecture community. This inclusivity allows for a broad spectrum of experimental activities, ranging from testing novel architectural optimizations to evaluating the impact of different hardware configurations on robotic systems performance.

Chapter 6

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

This chapter introduces an algorithm/hardware co-design for mobile robot path planning. 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 [196]. However, such offloading often cannot meet real-time requirements because the latency of communicating with the cloud is too high and unpredictable [196]. Thus, real-time applications 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 [253].

In this chapter, 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 [123, 182]. *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*, and their variants and extensions exhibit little to no parallelism for path planning [214]. 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 [229], they can be *semantically* predicted using domain knowledge. Specifically, as we show in §6.1.2, 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, *RACOD* 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 §6.7.1, prior work on hardware acceleration of collision detection [182,200] 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×.

Finally, in §6.6, we introduce an extension of *RASExp*, named RA*, which targets workloads beyond path planning.

Publications & Materials

The contributions of this chapter are published in [112, 118], and an implementation is available at:

https://github.com/cmu-roboarch/runahead-astar.

6.1 Motivation

6.1.1 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. An example is given in Figure 6.1 (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.

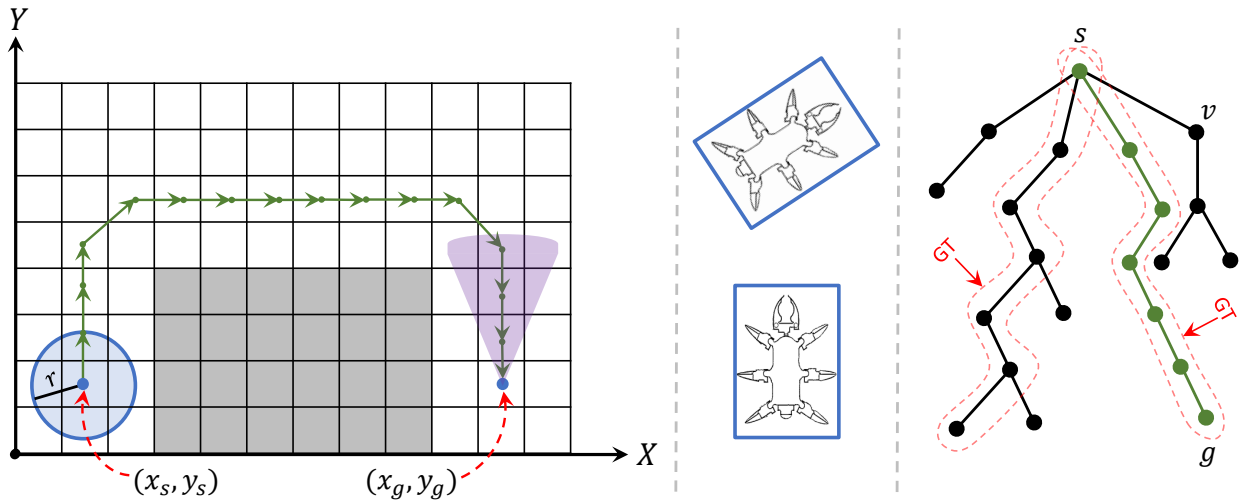


Figure 6.1: 2D mobile robot path planning (left), oriented bounded box (middle), and path graph search (right).

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 [4] is shown inside the rectangles in Figure 6.1 (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) [142] 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 6.1 (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 6.1 (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 6.1 (left) example, a 256-byte cache *dedicated* for collision detection memory accesses results in a 99+% hit ratio.

6.1.2 Path Search

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 inter-cardinal 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. As explained in §4.2.4 and §5.3.5, A* [158], along with its variants and extensions (§6.4.9), is the seminal algorithm widely used in various robot path planning applications. The key novelty of A* 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 §6.2.2).

Consider the graph depicted in Figure 6.1 (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* 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 work, the default heuristic is *Euclidean distance*.

A* 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* 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* path planning is serial, as are most of its extensions and variants [214, 231]. 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.

Patterns Exposed During Path Search

The green arrows in Figure 6.1 (left) show how the robot moves following the optimal path returned by A*. 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 6.1 (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 6.1 (left)). Figure 6.4 in §6.4.3 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 6.1 (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 6.1 (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* (§6.4.7).

6.2 Runahead Collision Detection (RACOD)

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

6.2.1 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.

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 [143]. 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 6.1.

Table 6.1: *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.

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 6.2 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 6.1), 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 (§6.2.1).

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 $(x_o + L, y_o + W, z_o)$

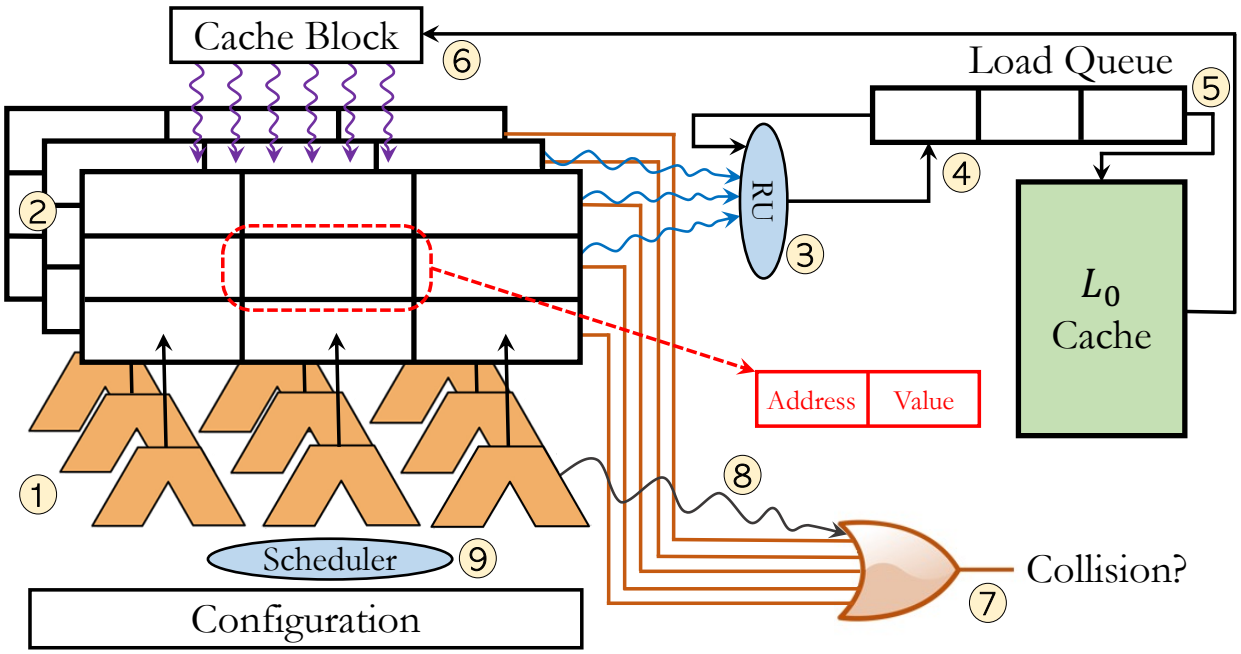


Figure 6.2: Hardware realization of *CODAcc*.

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*.

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. Notice, 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; the RU supports bit-granular, irregular (oriented) address coalescing while most GPUs coalesce *regular* addresses at a word granularity.

Noteworthy, due to the high spatial locality (§6.1.1), 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

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

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.

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.

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.

6.2.2 Run-Ahead State Exploration (RASExp)

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 (§6.1.2) 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 (§6.1.2).

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* and *RASExp*'s extension. The pseudo-code depicts a multithreaded baseline A*, 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 (§6.1.2). Then some basic operations of A* 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* 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*).

```
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* and *RASExp*'s extension.

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 work, 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.

6.3 Methodology

We evaluate our hardware accelerator alone and in conjunction with our algorithm extension, using simulation, in §6.4. We also evaluate a software-only implementation of our algorithm extension on commodity hardware (CPU and GPU) in §6.5.

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 [224], and model a processor after the Intel Core i3-8109U [105]—a state-of-the-art robotic processor deployed in LoCoBot [103]. We simulate all programs to completion.

We conduct our software-only evaluations using a 32-core Intel E5-2670 CPU [102] and an NVIDIA GeForce GTX 1060 GPU [107]. We use Ubuntu 18.04 with Linux Kernel 4.15 as our operating system.

6.4 Evaluation

6.4.1 Accelerator's Specifications

Table 6.2 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 6.2: 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: $10 \times 3 \times 3$ 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 [188]. Even tiny ARM Cortex-A15 cores in 40 nm technology occupy $4.5mm^2$ and consume $1W$ [188].

Due to its low overhead, we can integrate tens of *CODAcc*s with the processor. The area overhead of thirty-two *CODAcc*s altogether plus the cache extension (§6.2.1) 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).

6.4.2 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 [237]. Figure 6.3 (top) shows snapshots of the environments, and Figure 6.3 (bottom) shows *RACOD*'s speedup on them, varying the number of accelerators.

For every map, we choose 100 random start/goal points. The graph is 8-connected, and a multithreaded A* 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 *CODAcc*s.

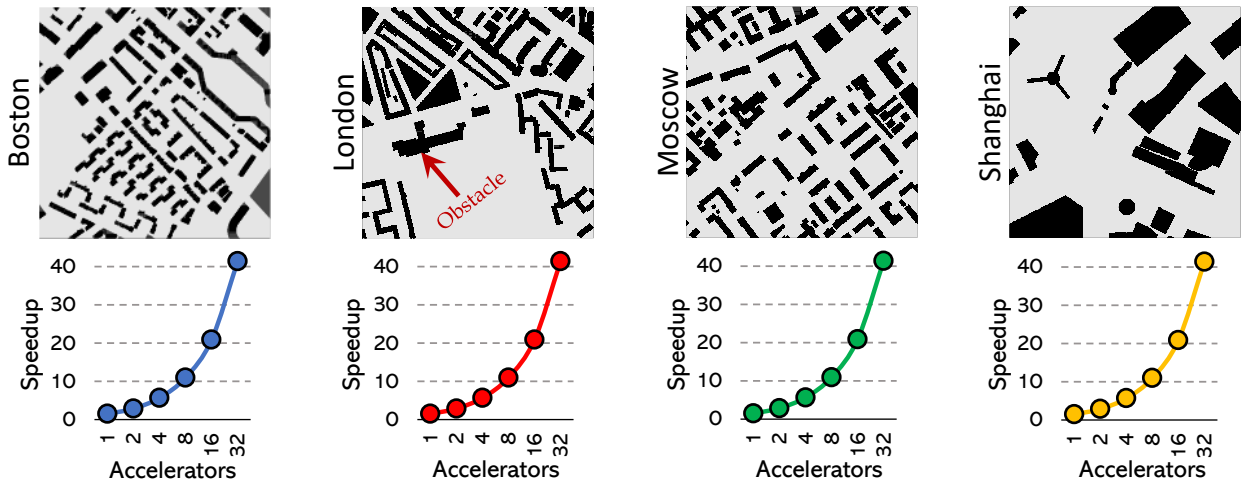


Figure 6.3: 2D navigation in the wild.

Interestingly, we observe similar *normalized* speedups with different maps. This mainly emanates from the high prediction coverage/accuracy of *RASExp* (see §6.4.7), which results in predicting *enough correct nodes* in all the environments and thereby effectively keeping all the accelerators utilized (see §6.4.8). As a result, *RACOD* brings *linear* speedup proportionate to *speculation runahead* across all the maps.

6.4.3 Exploration Footprint

Figure 6.4 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.

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 §6.4.7).

6.4.4 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' [23] as our environment. The resolution of the map is 0.2^m. Figure 6.5 shows the environment (left) and *RACOD*'s performance improvement with different numbers of accelerators (right).

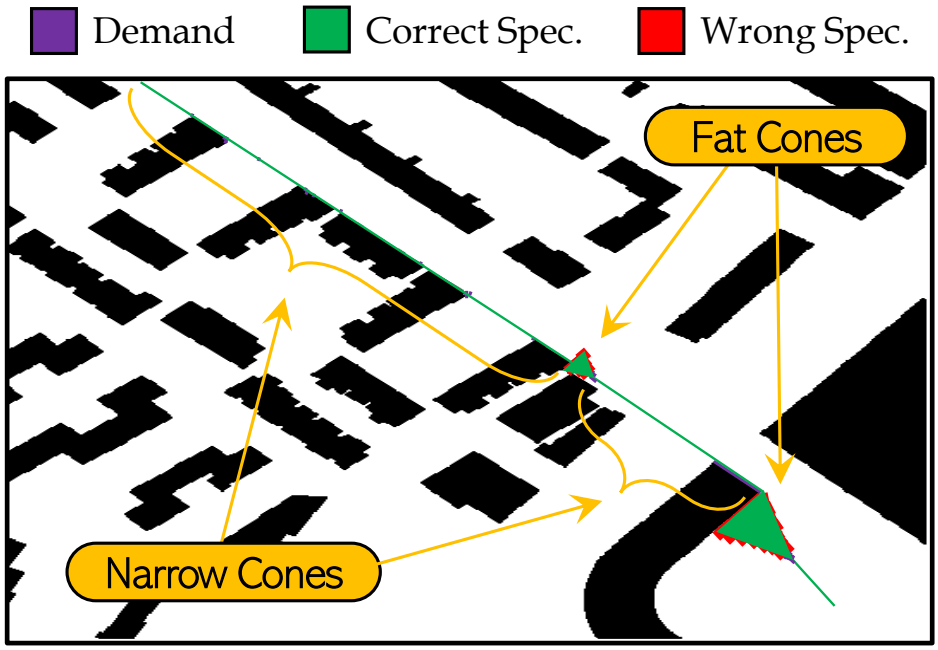


Figure 6.4: Cone-like patterns in a Boston snapshot, with a runahead of 32.

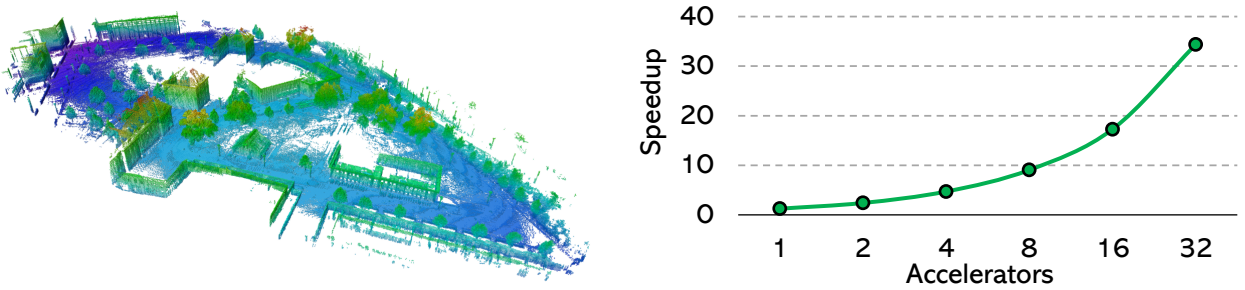


Figure 6.5: A map of the environment and the performance improvement.

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 *CODAcc*s.

6.4.5 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)* [179] is a widely-used robotic arm planning algorithm. The main advantage of RRT over former methods like PRM [166] 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 [103], operating in the environment shown in Figure 6.6 (left), planned by a state-of-the-art parallel RRT [178]. 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 6.6 (middle). Figure 6.6 (right) plots the speedup with 1–4 accelerators.

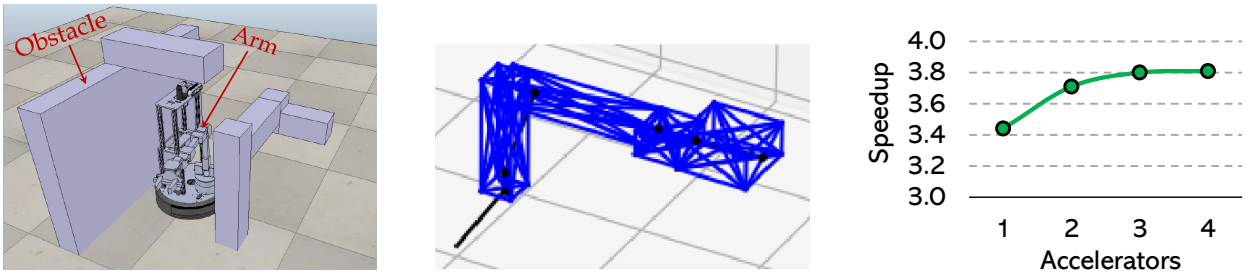


Figure 6.6: The modeled robot and environment (left), LoCoBot's *OBB*s (middle), and the performance improvement of hardware acceleration (right).

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×.

6.4.6 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 [143].

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. Copying latency is assumed to be captured in communication latency. Also, the communications have blocking semantics, meaning

the processor waits for the operation to finish before proceeding. Figure 6.7 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.

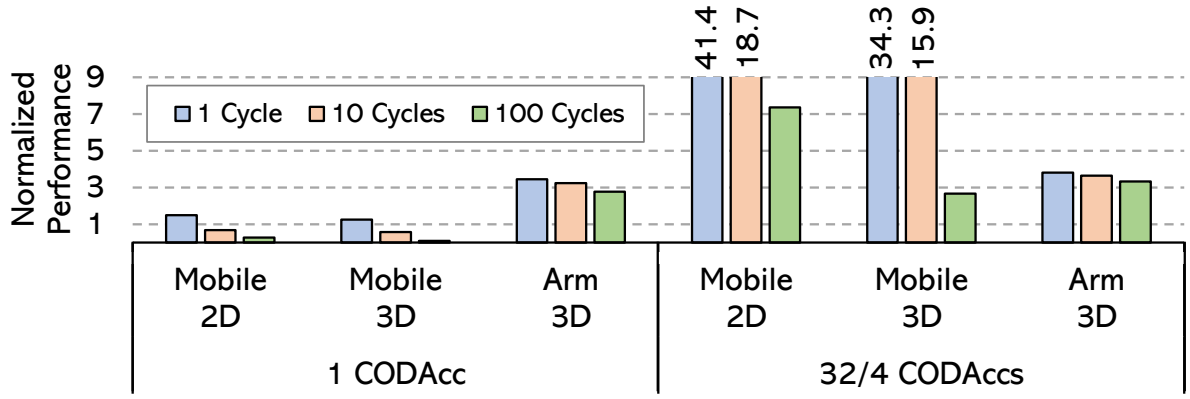


Figure 6.7: Speedup sensitivity to CPU-accelerator communication latency.

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.

6.4.7 Prediction Coverage and Accuracy

Semantic Predictor

Recall that *RASExp* uses semantic information and is implemented in software (§6.2.2). Figure 6.8 (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.

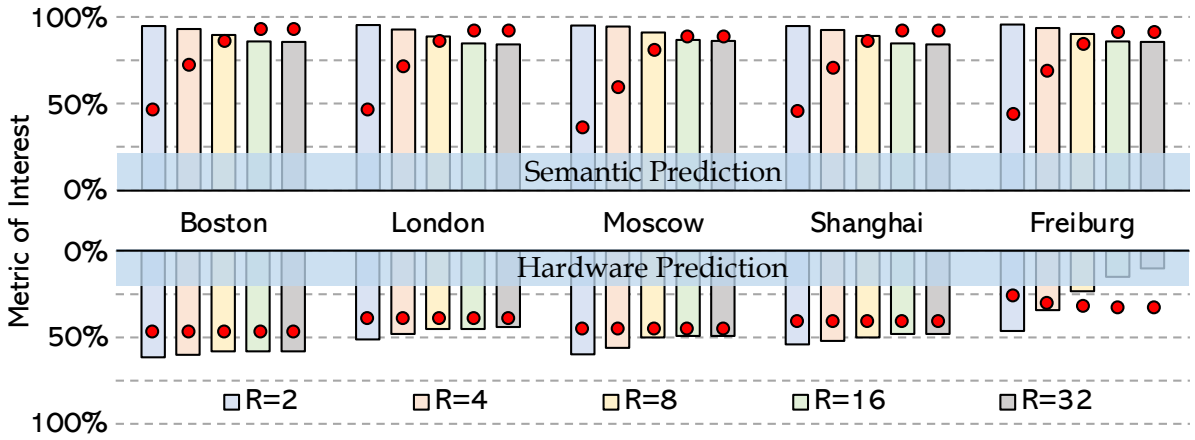


Figure 6.8: Prediction accuracy/coverage (bars/dots) with different runaheads.

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.

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).

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 6.8 (bottom), we study *RASExp* with a state-of-the-art hardware predictor.

We *repurpose* VLDP [229], 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$\times$ coverage and 2$\times$ 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 [119] 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 [117] would be meaningless in this context since collision detection sequences never repeat: there is at most one collision check per state.

6.4.8 Division of Labor

The bars in Figure 6.9 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.

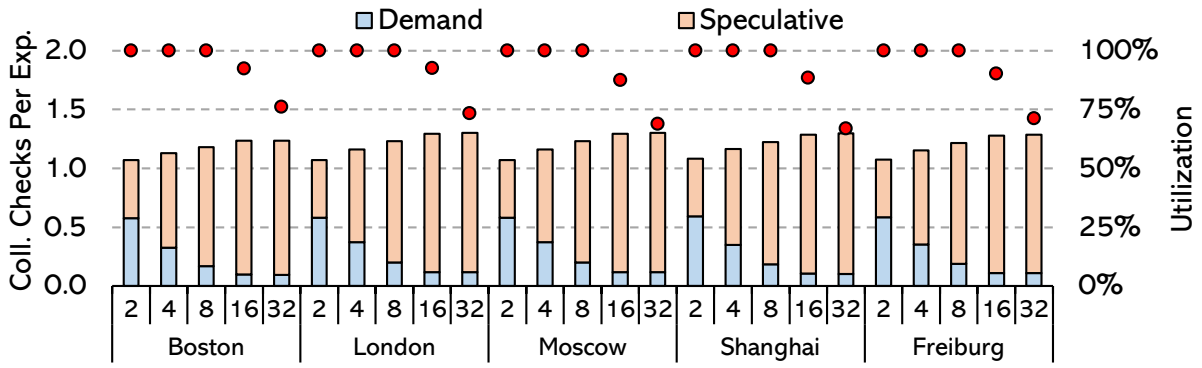


Figure 6.9: Division-of-labor, varying the number of accelerators.

The solid dots in Figure 6.9 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 (§6.2.2).

6.4.9 Weighted A* & Different Heuristics

A* is guaranteed to find the shortest path (*optimal*), with the minimum number of expansions. 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* (WA*) [216] is the most popular satisfying algorithm for heuristic search in various domains. WA* *inflates* the heuristic by a factor of $\varepsilon > 1$. That is, WA* 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 re-evaluate the experiments of §6.4.2 with two other popular 2D heuristics: *Manhattan distance* and *non-uniform diagonal distance*. We also evaluate the Dijkstra search algorithm, which does not use any heuristics.

Figure 6.10 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.

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×.

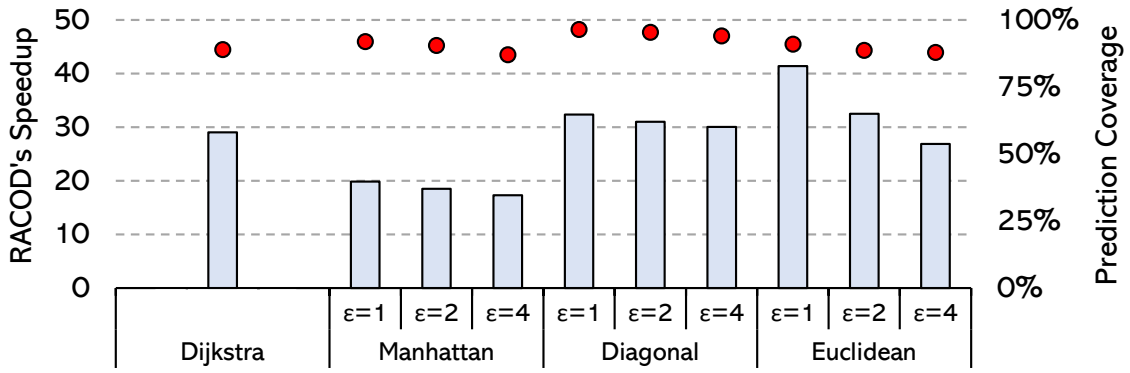


Figure 6.10: RACOD's effectiveness with WA* and different heuristics.

Also, Dijkstra is on average 25× slower than A*, and the performance of different heuristics is within 1.2×–5.3× of each other.

6.4.10 L_0 Cache Configuration

Figure 6.11 shows L_0 hit ratios with different sizes. As shown, a 256 B cache is sufficient to filter the majority of requests.

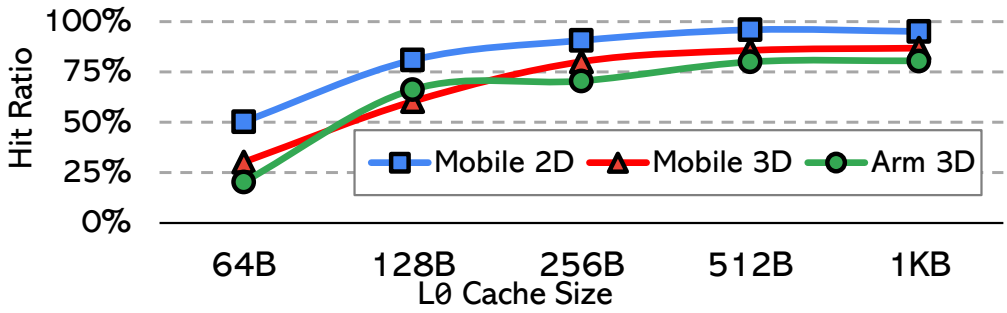


Figure 6.11: L_0 cache hit ratio with varying its size.

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.

6.4.11 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 (§6.4.2), in which, with a probability of 10%–70%, we inject *random* obstacles to an initially free space. Figure 6.12 shows how the prediction accuracy (left) and coverage (right) of *RASExp* vary. In this experiment, the runahead is 32.

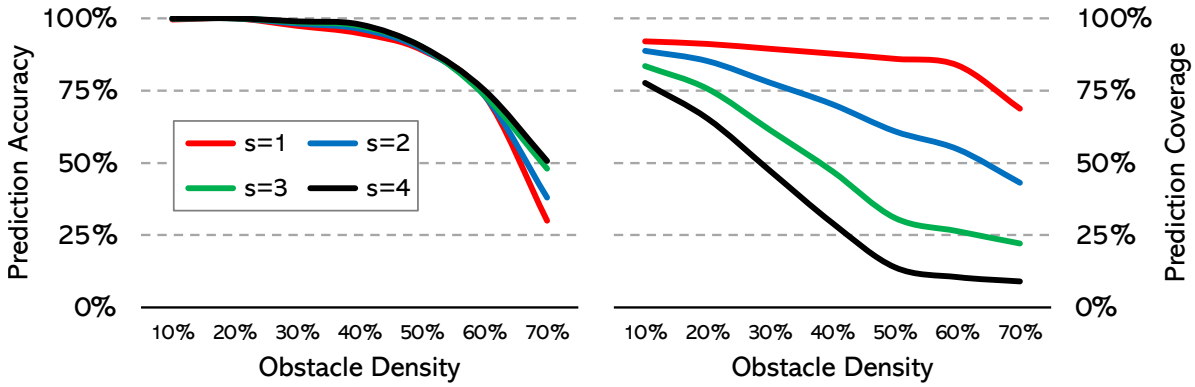


Figure 6.12: Prediction throttling impact with different obstacles densities, varying the predictor trigger threshold s.

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 §6.1.2, the organization of real environments is not so irregular that it would destroy patterns that emanate from geometric features of path planning.

6.5 Runahead Multithreading

Parallelizing path search algorithms like Dijkstra, A*, and WA* 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 [127, 172, 214] proposes methods for *safe* parallelization of expansions. For example, PA*SE [214] 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 6.13-(a, b) show the speedup of *(i)* A* with *Baseline Multithreading (BM)* (on expansion, all the node's eligible neighbors are evaluated in parallel), *(ii)* PA*SE [214], and *(iii)* A* 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.

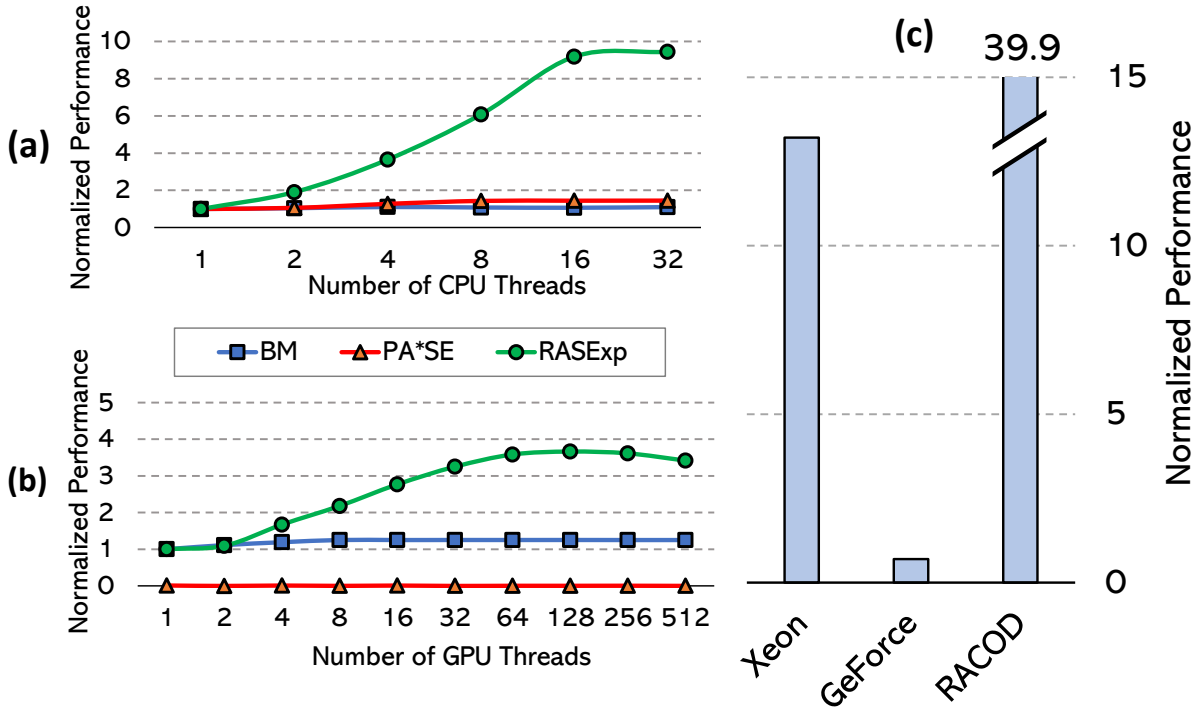


Figure 6.13: Performance comparison of different platforms and configurations. All but RACOD are commodity hardware.

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 (§6.1.2). 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 [214] were only up to 8 cores). *(ii)* The overhead of finding independent states is high; this issue is acknowledged in [214], but because the authors' evaluations were on a large PR2 robot [214] 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 6.13.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 [105], a typical processor used in modern robotic systems like the modeled LoCoBot [103]. 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 [105].

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.

6.6 RA⋆: Speculative Parallelism for A⋆ with Slow Expansions

In this section, we design and evaluate a *generalized* version of *RASExp*, named *Runahead* A⋆ (RA⋆). Unlike *RASExp*, which was specifically developed for robot path planning, RA⋆ aims at a broader spectrum of A⋆ algorithm applications. More precisely, RA⋆ is designed for "slow-expansion" scenarios in A⋆ applications where neighbor evaluations, such as validity checking and heuristic calculations, are time-consuming.

Furthermore, while *RASExp* was conceived based on empirical observations in path planning, RA⋆ is developed with analytical underpinnings, aiming to be applicable to a wider array of applications.

We evaluate RA* for five different applications of A*, namely robot path planning in (x,y), (x,y,θ), (x,y,z) domains, protein design, and symbolic task planning.

6.6.1 Definitions, Notations, and Scope

Following, we define some terms that we use throughout the section:

Evaluation: The process of *(i)* checking whether a state is valid (e.g., collision detection in robot path planning), and *(ii)* calculating its heuristic cost. In slow-expansion applications of A*, evaluations take the majority of execution time.

Pre-Evaluation: Evaluation done by RA*, ahead of time.

Expansion: The process of *(i)* picking the node with the lowest f from `OPEN`, *(ii)* checking whether it is the goal, *(iii)* marking it as 'visited' to avoid its re-expansion, and *(iv)* obtaining its neighboring nodes, evaluating them, and (conditionally) pushing them to `OPEN`.

Action/Direction: The operation that must be done to obtain a node from its predecessor node. In this section, we use the terms 'action' and 'direction' interchangeably.

Figure 6.14 shows a general example of an A* graph search. Every node represents a state with n possible actions. For example, in 2D path planning with a robot that can move in four cardinal directions, every node represents an (x,y) state, and every action represents a direction the robot can take (e.g., a_1: up, a_2: right, a_3: down, a_4: left). In this section, we assume that the maximum number of actions in every node is the same (e.g., $n = 4$ in the example). This assumption holds in a variety of applications of A*, including but not limited to the ones that we evaluate in this work (see §6.6.6).

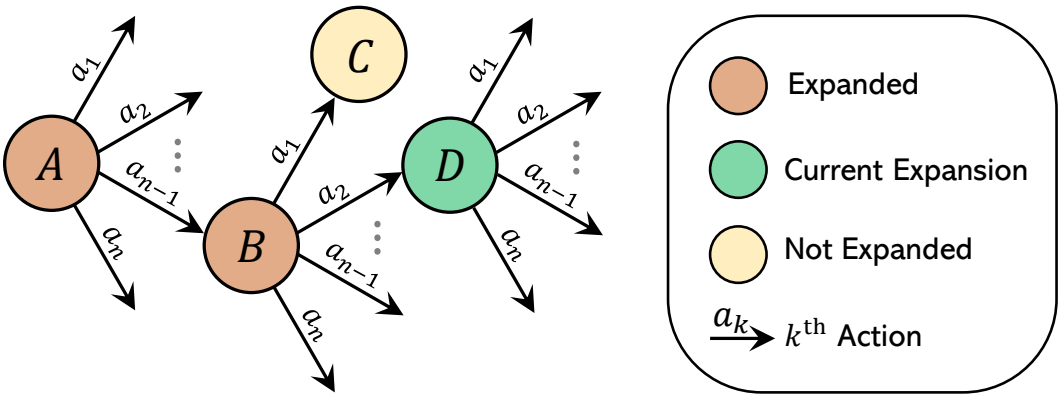


Figure 6.14: Graph search using A*.

Following, we define some notations that will help us explain RA*. The examples refer to Figure 6.14's graph.

- $pa(S,i)$: the i^{th} recursive *pa*rent of S.
  - ◇ Examples: $pa(B,1) = A$, $pa(C,1) = B$, $pa(C,2) = A$, $pa(D,1) = B$, $pa(D,2) = A$.

- ◇ The 0^{th} recursive parent of a node is the node itself: $pa(S,0) = S$.
- ◇ $pa(S,i) = pa(pa(S,j), i-j)$ holds for any valid $i \geq j$.

- $a(S)$: the preceding *a*ction of S. It is the action that led to obtaining S and adding it to `OPEN`.
  - ◇ Examples: $a(B) = a_{n-1}$, $a(C) = a_1$, $a(D) = a_2$.
- $a(S,i)$: the preceding *a*ction of $pa(S, i-1)$.
  - ◇ Examples: $a(B,1) = a_{n-1}$, $a(C,1) = a_1$, $a(C,2) = a_{n-1}$, $a(D,1) = a_2$, $a(D,2) = a_{n-1}$.

6.6.2 Runahead A⋆

6.6.3 The Abstract Predictor

In this section, we explain RA⋆ with an *abstract* predictor. We defer the explanation of the evaluated predictors to the next section.

The first operation RA⋆ performs upon an expansion is predicting *which successor of the expanded node is more likely to be expanded in the future*. We call it the Most Likely Successor (MLS). Note that, MLS is *not* necessarily the very next expansion in time order; MLS is the next expansion whose parent is the current expansion.

RA⋆'s predictor exploits correlations among previous preceding actions of an expanded node to predict the preceding action of its MLS (to reiterate, this is the action that leads to obtaining MLS and adding it to `OPEN`). Once the preceding action is predicted, the MLS can be trivially computed.

The prediction problem can be viewed as a classification problem where every action is a *class*, and the learning task is to learn the probability of an action *Act* being the preceding action of MLS in state S, given the history of past actions (Eq. 6.1). And, the prediction goal is finding the action that maximizes this probability (Eq. 6.2).

$$P(Act | a(S,1), a(S,2), a(S,3), \dots) \tag{6.1}$$

$$\underset{i \in \{1,\dots,n\}}{\arg\max}\, P(a_i | a(S,1), a(S,2), a(S,3), \dots) \tag{6.2}$$

For example, in Figure 6.14's graph, if a_k is the preceding action of D's MLS, we can write Eq. 6.1 as $P(a_k | a_2, a_{n-1})$.

6.6.4 The Design of a Predictor

As explained, upon expansions, RA⋆ predicts MLS by predicting its preceding action, based on the history of past preceding actions (i.e., Eq. 6.2). Ideally, the correlations among preceding actions can be learned using a recurrent neural network (RNN), where past actions $(a(S,1), a(S,2), a(S,3), \dots)$ are input features, and the future action (Act) is the model's output label. While such a scheme could offer a high level of prediction accuracy, we find that, in practice,

its high *prediction latency* (i.e., the time it takes to generate a prediction) presents challenges to RA*, downgrading its performance.

In this section, we explore simple, practical prediction approaches to achieve low prediction latency while offering a decent level of accuracy.

Straight-Line Predictor

Our proposal for making practical predictions is what we call Straight-Line Predictor (SLP). Upon expanding S, SLP predicts that MLS's preceding action will be $a(S)$. In other words, SLP simplifies Eq. 6.2 to Eq. 6.3,

$$\underset{i\in\{1,\ldots,n\}}{\arg\max}\, P(a_i|a(S)) \tag{6.3}$$

and considers that argmax i is such that $a_i = a(S)$. For the different applications that we consider, we found that $a(S)$ is a good approximation to the solution of the argmax problem.

SLP, in fact, serves as the predictor that *RASExp* employs to predict future directions within the context of mobile robot path planning.

Figure 6.15.a is a motivating example for SLP. It depicts 2D path planning for a mobile robot moving from state S to state G. The planner's purpose is to find the shortest path, and it uses the consistent heuristic function $h(S) = \|G - S\|_2$ (Euclidean distance).

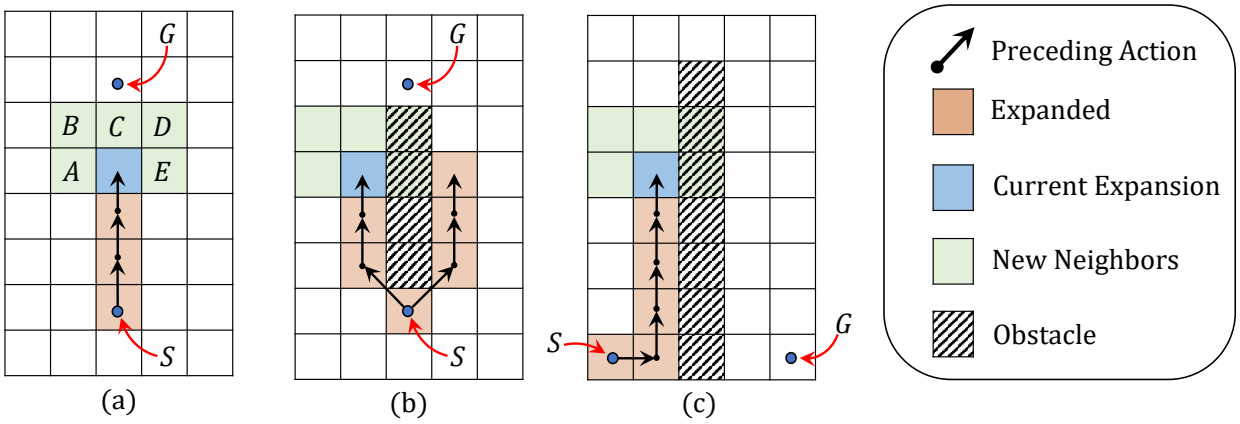


Figure 6.15: 2D path planning example.

Intuitively, to traverse the shortest path, the robot should head right towards the goal and *keep moving in the same direction* until either hitting an obstacle or reaching the goal. This intuition, in fact, emanates from the geometry principle that the shortest distance between any two points in an obstacle-free plane is a straight line.

Mathematically, after the current expansion, state C is expanded before $\{A,B,D,E\}$, given that C is not occupied. In other words, the next action after the current expansion will be the same as the preceding action—the premise of SLP.

SLP *mispredicts* when it hits an obstacle (generally, an invalid state). One might think SLP would work poorly in real-world environments where there may be many obstacles. However, obstacles in real environments are not so irregular that they destroy SLP's premise: SLP still offers decent prediction accuracy even in crowded environments.

For example, consider Figure 6.15.b and Figure 6.15.c, where the environments have obstacles. While moving from free space towards the goal, the direction changes upon hitting the first occupied space, which costs SLP a misprediction. However, *after that, preceding directions remain stable*, which helps SLP make correct predictions. This happens because occupied cells are not randomly scattered in the area; they are largely *co-located*, representing a (large) object (e.g., a wall).

Figure 6.15.b also shows how SLP deals with "temporally-interleaved" expansions, i.e., when consecutive expansions (in time order) belong to largely separate subgraphs. In Figure 6.15.b, A* (alternately) expands nodes on the two sides of the long obstacle; consecutive expansions in time order belong to largely separate subgraphs. When so, upon an expansion, the very next expansion in time order is *not* a child of the current expansion; however, the next but one expansion is. The intuition of SLP is that the predicted child, if correct, will eventually be expanded; no matter when a node gets expanded, its pre-evaluation will speed up the execution at that point.

We will show that SLP effectively predicts expansions in applications even *beyond path planning*. For example, in protein design, where the goal is minimizing an energy function, expansions follow straight-line patterns: if an action has resulted in (the largest) reduction in energy, repeating the same action is likely to reduce energy again. We will quantify the prediction accuracy in different applications and discuss when and when not patterns are predictable.

Random Action Predictor

For reference, we also evaluated a Random Action Predictor (RAP). RAP picks one of the possible *n* actions randomly. The purpose of including RAP is to compare it with SLP and shed light on the influence of prediction performance on the end-to-end speedup.

Sequitur

For reference, we also include Sequitur (SEQ) in our prediction evaluations. SEQ is originally proposed as a data compression algorithm [204]; nevertheless, later work [117] uses it to measure the *prediction opportunity*. The purpose of including SEQ is to have an upper bound on prediction performance, and evaluate how close SLP is to an ideal predictor. Note that SEQ works by offline processing of data (i.e., preceding actions in this work) and is not practical.

Figure 6.16 shows SEQ's operation with an example input stream depicted using symbols ('w', 'x', 'y', and so on). We take this example from [248]. In the context of RA*, the input stream is the sequence of preceding actions. SEQ processes the entire stream and builds a grammar whose production rules are formed such that it captures repetitions in the stream, as shown in the figure.

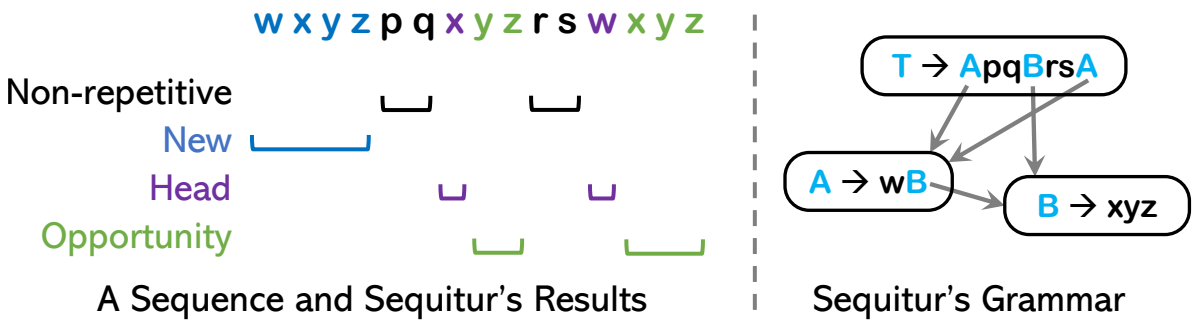


Figure 6.16: Sequitur's grammar with an example.

Using the formed grammar, SEQ classifies the symbols of the input stream into four groups: *(i) Non-repetitive* refers to symbols that never recur in the stream, and hence are unpredictable using any repetition-based prediction method; *(ii) New* refers to the first occurrence of symbols in the stream (i.e., before the predictor can learn them); *(iii) Head* refers to the first symbol of every rule, which cannot be predicted itself, but it can be used to identify and predict the rest of the symbols in the rule; *(iv) Opportunity* refers to the rest of the symbols.

Sequitur argues that *Opportunity* is what an ideal predictor can predict by learning the repetitions in the data stream. Hence, processing Sequitur's grammar can provide an upper bound on prediction performance. In the provided example, for instance, the best-performing prediction mechanism can predict up to $\frac{5}{15}$ of the symbols. An implementation of Sequitur processing is available at: https://github.com/bakhshalipour/SequiturAnalysis

6.6.5 Evaluation Methodology

Setup

We conduct our evaluations on up to 16 cores of Intel Xeon Gold 5218R processor [41]. RA⋆'s runahead is set such that it utilizes all the available cores. Our operating system is Debian 10, and our compiler is GCC 11. We use C++17 Thread Pool library [232] for threading operations. The default predictor of RA⋆ is SLP.

Applications

We evaluate five applications of A⋆, namely path planning in (x,y), (x,y,θ), (x,y,z) domains, protein design, and symbolic planning. Excluding symbolic planning, the expansions in these applications are slow.

Path Planning: For (x,y) and (x,y,θ) planning, we use Boston map from Moving AI [237]. For (x,y,z) planning, we use Freiburg map [23]. The heuristic function of all the planners is Euclidean distance. The (x,y) and (x,y,z) planners can move in 8 and 6 directions, respectively. In (x,y,θ) planning, the discretization granularity of θ is $\frac{\pi}{4}$. The agents in (x,y) and (x,y,z) are rectangle-shaped, and the agent in (x,y,θ) is a polygon. For all path planning scenarios, we choose ten random start-goal points, and report the results for the average of ten executions. We model

the other details of (x,y) and (x,y,z) planning respectively based on `pp2d` and `pp3d` of *RTRBench* (see §4.2.4 and §4.2.5).

Protein Design: As the input set, we use `1I27_A_H` from [255]. We use the heuristic function and the implementation of [256] for estimating the energy and modeling the details.

Symbolic Task Planning: We evaluate a symbolic planner solving the Blocksworld problem with eight blocks with a random initial organization. The task planner uses symbol differences as the heuristic function. We model the other details based on `sym-blkw` of *RTRBench* (see §4.2.11).

Competitors

We compare RA* against PA*SE [214] and HDA* [172]. PA*SE is a state-of-the-art parallel A* algorithm proposed for slow-expansion applications. PA*SE, unlike RA*, expands multiple nodes in parallel. More specifically, PA*SE discovers *independent states* and parallelizes their expansions. s and s' are two independent states if the expansion of s cannot lead to a shorter path to s', and vice-versa.

HDA* assigns nodes to processors (cores) using a hash function. It maintains multiple `OPEN` lists, one per processor. When a processor expands a node, instead of pushing all neighbors to its own `OPEN`, HDA* hashes each neighbor to determine which processor should process it. That neighbor is then sent to the determined processor and gets processed by it. By doing so, it parallelizes multiple expansions; however, it loses A*'s guarantee that every node is expanded at most once. This implies that when the goal is expanded for the first time, HDA* has not necessarily found the shortest path to it. Hence, HDA* might expand nodes multiple times to ensure the goal is achieved by the least-cost path.

Spurred by the observation that re-expansions can happen too many times, we make one modification to the original HDA*. Namely, after performing a validity checking for a node (e.g., collision detection), we store the validity status (e.g., collision or free) in a global hashmap. When a node gets re-expanded, we query the hashmap to get the validity status. This way, costly operations like collision detection are performed only once when a node is generated for the first time; after that, the memoized results are used. This technique makes re-expansions cheaper, but not free (see §6.6.6).

6.6.6 Evaluation Results

Execution Time

Figure 6.17 shows the speedup of the parallelization methods. The baseline to which the speedups are normalized is a multithreaded implementation of the vanilla A*: only neighbors of the expanded nodes are evaluated in parallel.

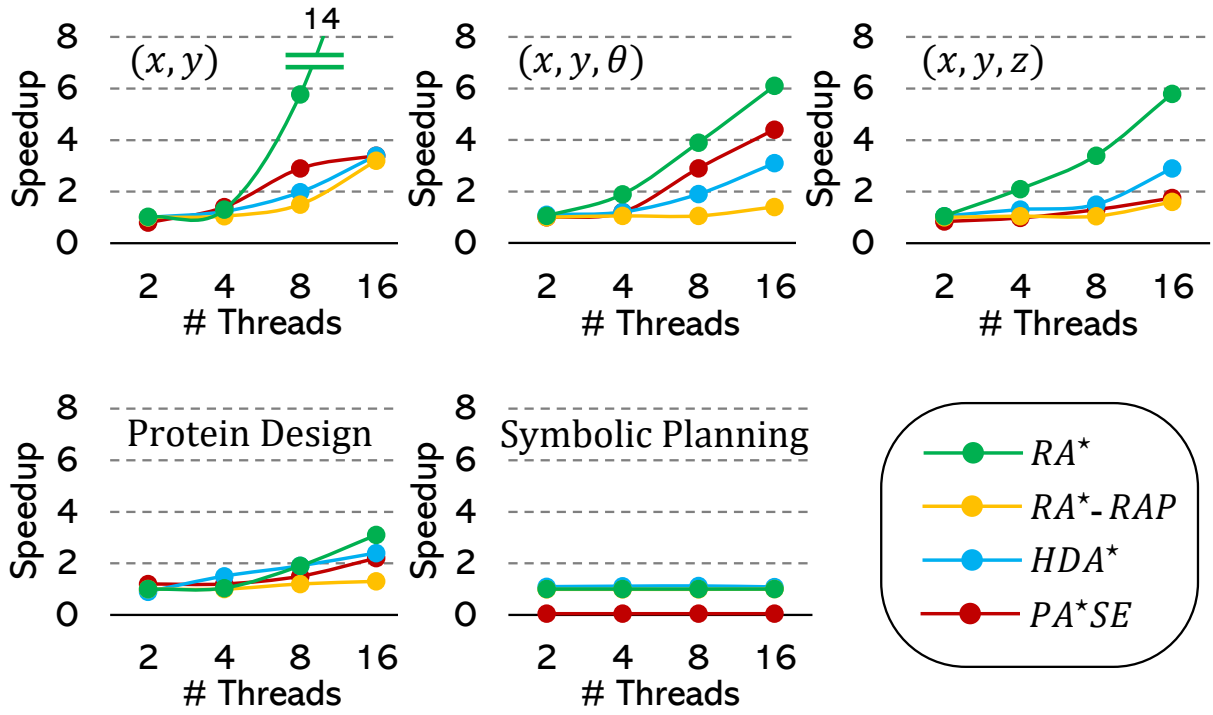


Figure 6.17: Speedup comparison of methods.

When the number of threads is small (e.g., < 4), the baseline A* utilizes most of the threads, leaving few free threads for RA*. Consequently, RA* does not have a considerable opportunity in such settings to improve performance. However, when the number of threads is large, RA* offers significant speedups in slow-expansion applications. Across them, RA* with 16 threads offers 3.1×–14.1× speedup over the baseline, outperforming PA*SE by 1.4×–4.1× and HDA* by 1.3×–4.1×. RA*'s high prediction performance, which we will present, is the main contributor to its large speedup. The importance of accurate prediction can be seen by comparing RA* (with its SLP predictor) to RA*-RAP (where a random predictor is used instead of SLP).

RA*, nonetheless, fails to accelerate the symbolic task planner, a fast-expansion program. The reasons are *(i)* the evaluations are not so costly that the parallelism benefits could outweigh the threading overheads, *(ii)* RA*'s predictions are not accurate in this domain, and *(iii)* the baseline vanilla A* almost always uses all the available threads to evaluate a large number of neighbors (the degree of graph vertices is significantly larger than the available threads), leaving few free threads for RA*.

We find that PA*SE suffers from two major issues. First, there are not always many independent states. This limits the amount of parallelism as the algorithm cannot find enough independent states to utilize all the available threads. Especially, when the number of threads is large (e.g., 16), the thread utilization can drop to even below 25%; this happens too frequently in applications in which the nodes in `OPEN` have close g values, and hence, they cannot pass PA*SE's "independence checks." Meanwhile, as we show in §6.6.6, RA* is often able to keep all the threads highly

utilized.

The second problem with PA*SE is the high overhead of discovering independent states. PA*SE repeatedly checks *all* the states in `OPEN` and in its `BE` (which tracks *being expanded* nodes) to discover independent states. This imposes significant overheads proportional to the number of nodes in `OPEN` and `BE` at every check, which could be many. For example, in protein design, 53% of the entire execution time (averaged across all threads) is spent discovering independent states.

On the other hand, HDA* suffers from other types of overheads. HDA* allows the re-expansion of nodes to account for the fact that states may get expanded before they have the minimal cost from the start state. The re-expansions can happen exponentially-many times, imposing a massive extra workload. For example, in (x,y) planning, HDA* increases the number of expansions by an average of $10.7\times$. Notice, memoizing the validity status that we add to HDA* (see §6.6.5) makes re-expansions significantly cheaper, but not free. Re-expansions still need to update HDA*'s data structures and send messages to the other cores; when re-expansions are too many, their cumulative overhead becomes significant.

The other performance-limiting components of HDA* are (P1) long waits at synchronization barriers, and (P2) long path reconstruction time [246]. P1 slows down execution, especially when the number of threads is large. For example, with two cores, only one core would ever wait for the other. However, with 16 cores, 15 of them could be waiting for 1 core to complete processing, even if every one of those 15 cores has work added to its `OPEN`. This wait time downgrades performance. P2 is a serial process with many send-and-receive messages. Throughout this process, the goal vertex's owner core should send and receive messages for every vertex in the shortest path whose owner is different. With more cores, it is more likely that vertices on the shortest path belong to other owners, and hence, more send and receive messages must be communicated.

We also found synchronization overheads non-trivial in both PA*SE and HDA*. PA*SE locks all insertions to and deletions from its heavily-used data structures like `OPEN` and `BE`. HDA*, when running on a shared-memory system, needs to lock cost and parent lookup tables for every vertex [246]. In contrast, RA* only needs to lock the hashmap of results[2] when it inserts evaluation and pre-evaluation results. As such, synchronization overheads are significantly lighter in RA*.

Prediction Performance

Figure 6.18 shows the prediction coverage and accuracy of different predictors. Recall that SEQ is impractical. Its accuracy is always 100%, and it makes a varying but *pattern-dependent* number of predictions per expansion; in

[2]The lock is implementation-dependent. For example, with a bucketized hashmap, only the manipulated bucket must be locked, not the entire hashmap; or, with a state-space-sized array as the hashmap, the lock is not needed at all, because there is a 1:1 mapping from states to array elements.

contrast, SLP and RAP make R predictions, where RA*'s runahead parameter R depends on the number of free threads in the system. The purpose of including SEQ is to gauge how practical predictors compare to an ideal predictor.

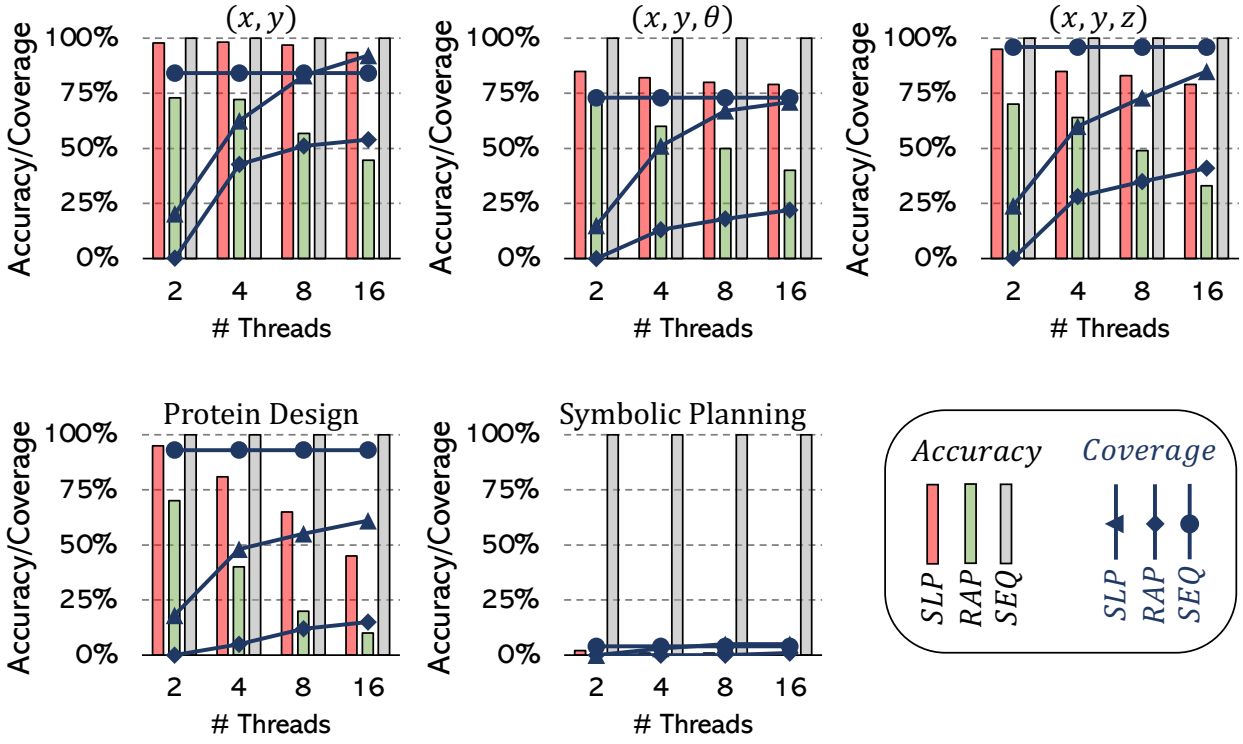


Figure 6.18: Performance comparison of different predictors. The default RA* predictor is SLP. SEQ is offline (impractical).

With increasing runahead (achieved by increasing the number of threads), RA* makes more predictions per expansion. By increasing the number of predictions, generally, the prediction accuracy drops and the coverage rises. The accuracy drops because it becomes less likely for a pattern to remain stable (e.g., a preceding action keeps repeating) for a larger number of iterations. On the other hand, the coverage increases as RA* runs *further ahead* and pre-evaluates more nodes.

In slow-expansion applications (i.e., all but symbolic task planning), with two threads (i.e., up to one prediction per expansion), SLP's accuracy is 85%–97.9%, which is 15%–25% higher than RAP and is within 2.1%–15% of SEQ. Interestingly, RAP's accuracy is far higher than $\frac{1}{n}$. We found that, in many cases, though RAP mispredicts the MLS, the *mispredicted MLS shares some neighbors with the actual MLS*; some of the pre-evaluations are eventually used. However, this is true when the runahead is small; with larger runaheads, RAP's predictions progressively diverge from the correct path. As a result, with large runaheads, RA* with RAP significantly underperforms RA* with the proposed SLP, as shown in Figure 6.17. With 16 threads, SLP's accuracy is 45%–93.5%, which is 35%–48.9% higher than RAP and is within 6.5%–55% of SEQ.

Also, with 16 threads, SLP offers 61.2%–92.2% coverage, which is 38.2%–49% higher than RAP and within an

average 7.4% proximity of SEQ. In (x,y), SLP's coverage is even up to 8% better than SEQ; however, this higher coverage comes at the cost of 6.5% less accuracy caused by aggressively making many predictions at once when there is a high number of free threads.

In symbolic planning, the predictors offer very poor performance. SLP fails because its postulations do not hold in the application; RAP fails because the number of actions (i.e., node degree) is large. SEQ offers only slightly better prediction coverage.

Note that mispredictions, when they are numerous, can waste power; threads perform computations whose results are not used. Importantly, the waste happens only at the computation level (CPU) and not the entire system (memory, robot's mechanics, etc.). Hence, the overhead can be neglected in many settings like with mobile robots where the *entire* computation contributes to $< 5\%$ of the total power consumption [124], or in the cloud where components like network and memory consume the majority of power.

Harnessing SLP

Our prediction performance results showed that SLP effectively predicts expansions in real-world path planning applications (and even beyond, i.e., protein design). However, one can indeed envision (synthetic) environments in which SLP is misled, exhibiting poor accuracy. In settings where the prediction accuracy is of significant importance, e.g., processors with few cores in which the system cannot afford to waste threads doing useless computation, this could lead to performance degradation. We propose a supplementary technique to "harness" SLP in such cases.

We speculate only if the path leading to expansion was *stable* in the last s steps; i.e., $a(S,1) = a(S,2) = \ldots = a(S,s)$. For example, with $s = 3$, speculation runs if the expanded node's preceding action is the same as its parent's preceding action, and its parent's parent's preceding action.

To evaluate this mechanism, we generate synthetic 2D maps (like the evaluated Boston) for (x,y) planning. The maps are initially empty; with a probability of p, we add *random* obstacles to them. Figure 6.19 shows SLP's prediction performance with 16 threads and different p and s. The bars show prediction accuracy and the lines show prediction coverage.

As the results show, the harnessing mechanism effectively regulates the predictor's aggressiveness: with $s = 4$, it harnesses the aggressiveness such that even in a 70%-occupied environment with random obstacles, the accuracy is still above 72.1%. On the flip side, the coverage drops as a result of reduced prediction opportunities.

Another point revealed by this experiment is the large difference in the prediction performance of SLP in realistic and random environments. For example, with $s = 1$, SLP offers 39.3% more accuracy in Boston compared to the 70%-random map, and also, it covers 10.1% more evaluations. Our results confirm that path planning in real-world environments exhibits predictable patterns. To illuminate this, Figure 6.20 compares SLP's performance in multiple

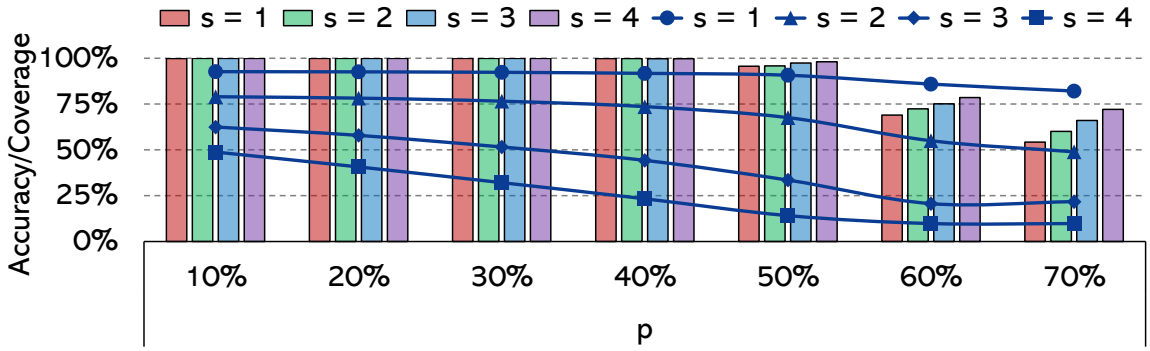


Figure 6.19: Harnessing SLP on random maps.

cities' maps of Moving AI [237] with the 70%-random map. The bars show the accuracy and the dots show the coverage.

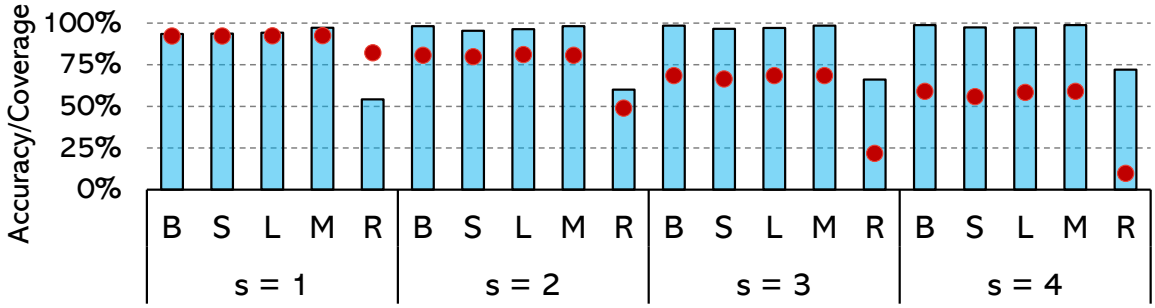


Figure 6.20: SLP in real-world (Boston, Shanghai, London, Moscow) and random (70%-Random) environments.

With $s = 1$, the worst-case accuracy/coverage of SLP in the real-world maps is 39.3%/10.1% higher than in the random environment. With $s = 4$, it is 25.3%/46% better.

Planning Resolution

To study the impact of *evaluation time* on RA*'s speedup, we coarsen the resolution in (x, y) planning. With such coarsening, the *number* of obstacles decreases, because multiple nearby obstacles are considered one big obstacle. As a result, collision detection becomes faster; on the downside, the free space is underestimated. The bars in Figure 6.21 show RA*'s speedup with different numbers of threads and different coarsening factors. A coarsen factor of M means that the planner considers M times fewer units (pixels) in every map dimension. For example, a 16× coarsening means the considered map has 16×16 times fewer units than the original map. The blue line with solid dots in the figure shows the average evaluation (collision detection) time with the corresponding resolution.

The results show that RA* offers considerable speedups with different resolutions. It is expected that with decreasing evaluation time, the speedup provided by parallel methods becomes smaller (see §6.6.7), as a result of increasing the *fraction* of time the program spends in the serial part (e.g., manipulating `OPEN`). This expectation generally holds

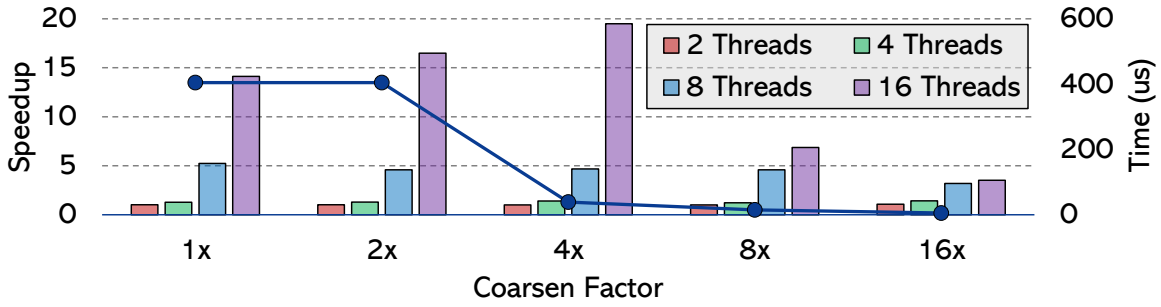


Figure 6.21: RA*'s speedup with coarsening the resolution.

true for RA* with a few anomalies that happen because of taking different execution paths; i.e., with different coarsening factors, the planning is essentially done on *different maps*, which results in taking different execution paths. When the collision detection is costly ($38\mu s - 404\mu s$), RA* offers 14.1×–19.5× speedup; when the collision detection becomes faster ($5\mu s - 14\mu s$) its speedup drops to 3.5×–6.9×.

Weighted A*

Weighted A* (WA*) [216] inflates the nodes' heuristic cost by a factor of $\varepsilon > 1$. Doing so, nodes are expanded in the order of $f = g + \varepsilon \times h$. This way, the search is biased towards the nodes that are closer to the goal. With WA*, the search terminates earlier (i.e., the goal itself is expanded sooner), but the solution could become ε times costlier.

Figure 6.22 shows the speedup of the methods for WA* with $\varepsilon = \{1,2,4\}$ for the average of slow-expansion applications. As the results show, RA* consistently outperforms PA*SE and HDA*. However, with increasing ε, the speedup of RA* drops, because of the reduced opportunity; the larger ε, the fewer expansions.

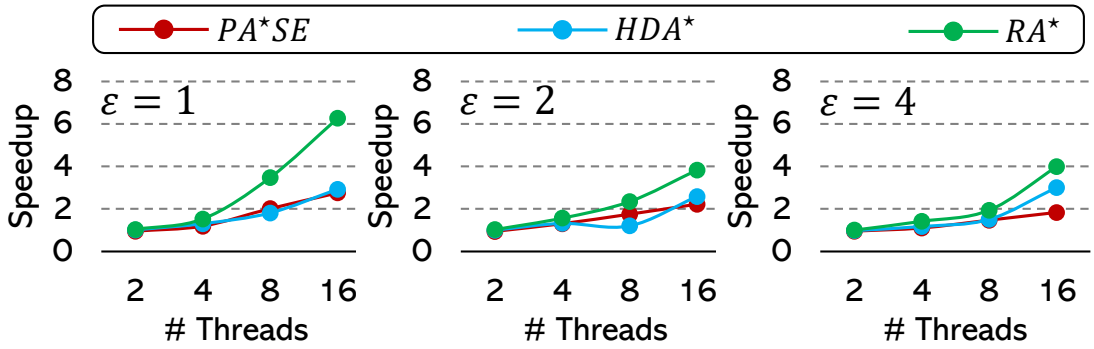


Figure 6.22: Speedup comparison of methods for WA*, averaged over the slow-expansion applications.

RA* on Top of Prior Work?

Theoretically, RA* can be orthogonally used with PA*SE and HDA*, as well as many other prior approaches. RA* is orthogonal to parallel best-first search algorithms because those algorithms try to expand multiple nodes in parallel; in

contrast, RA* runs ahead of individual expansions and paves the way for accelerating future expansions; they exploit different sources of parallelism.

Nevertheless, we observed that combining RA* with prior approaches does not improve performance significantly. Not unexpectedly, combining these approaches results in stacking their overheads; unfortunately, the increased parallelism does not outweigh the increased overheads. Particularly, since HDA* relies on many node-to-node communications, its overheads are entirely stacked with the overheads of RA* and results in slowdown rather than speedup. We leave to future work the exploration of an efficient combination of RA* with other parallel algorithms.

Workload Distribution and Thread Utilization

Bars in Figure 6.23 show the average number of useful (pre-)evaluations (i.e., evaluations and pre-evaluations whose results are ultimately used) per node expansion. The bars are broken down into demand and speculative; demand represents the evaluations done by the baseline algorithm, and speculation represents the pre-evaluations of RA*. With increasing the number of threads, the fraction of speculative evaluations increases. This way, more evaluations are lifted from the critical path, reducing the execution time.

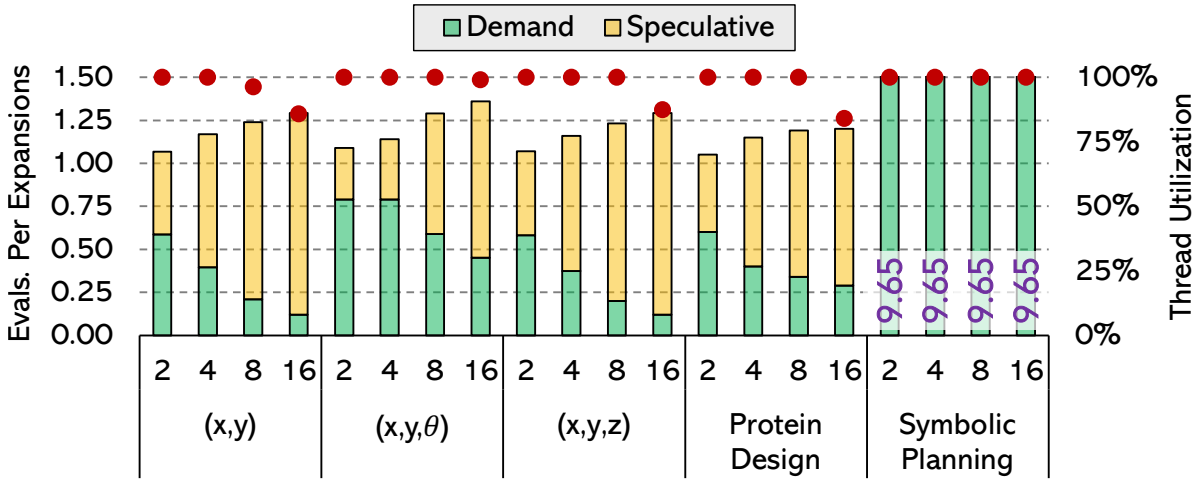


Figure 6.23: Workload distribution and thread utilization with different numbers of threads.

The solid dots in Figure 6.23 show the thread utilization. We define thread utilization as the fraction of threads performing evaluations over all threads in non-idle expansions (i.e., expansions with at least one evaluation). When the total number of threads is fewer than 8, the utilization ratio is almost 100%. This shows that with RA*, there is enough workload to keep all threads busy. With increasing threads, the utilization ratio starts decreasing because of the finite nature of the parallelism even in the presence of speculation. With 16 threads, the utilization ratio is 84%–99% in slow-expansion applications.

6.6.7 Speedup Projection

We develop an analytical model for RA* to project its speedup beyond the studied applications. We recall that RA* does not expand any nodes during the speculation phase. Hence, let us assume both the baseline A* and RA* expand N nodes in total. Let us assume every node has n neighbors, and the system supports the parallel execution of $t > n$ threads. Consider t_{serial} as the execution time of the serial portion of the algorithm at every expansion (i.e., everything other than neighbor evaluation), $t_{threading}$ as the threading overhead (i.e., the time it takes to manage all threads), and t_{eval} as the evaluation time. The total execution time of the baseline can be formulated as:

$$T_{Baseline} = N \times (t_{serial} + t_{threading} + t_{eval}) \tag{6.4}$$

Note that we assume all neighbors are evaluated in parallel, as there are more threads available than the number of neighbors (i.e., $t > n$). Now, we formulate the execution time of RA*. Let us assume P_{cov} is the fraction of correctly predicted expansions out of all expansions, i.e., prediction coverage; and, P_{acc} is the prediction accuracy.

Also, consider l as the prediction latency, and R as the runahead. In non-idle expansions (i.e., expansions with at least one evaluation), RA* pre-evaluates R nodes; it predicts $\frac{R}{n}$ nodes and pre-evaluates their (up to) n neighbors. We call $R' = \frac{R}{n}$ *predictor runahead*.

At every expansion, with a probability of P_{cov}, no evaluation is performed (the memoized results are loaded). Consider t_{load} as the time it takes to load the memoized results. We can formulate the execution time of RA* as:

$$\begin{aligned} T_{RA^\star} = N \times (t_{serial} + P_{cov} \times t_{load} \; + \\ (1 - P_{cov}) \times (t_{threading} + t_{eval} + R' \times l)) \end{aligned} \tag{6.5}$$

As mentioned, l is the latency of generating one prediction, and hence, $R' \times l$ is the latency of generating all predictions.

Considering that in non-idle expansions, RA* evaluates n demand nodes and $R = R' \times n$ speculative nodes, we can approximately[3] formulate P_{cov} as:

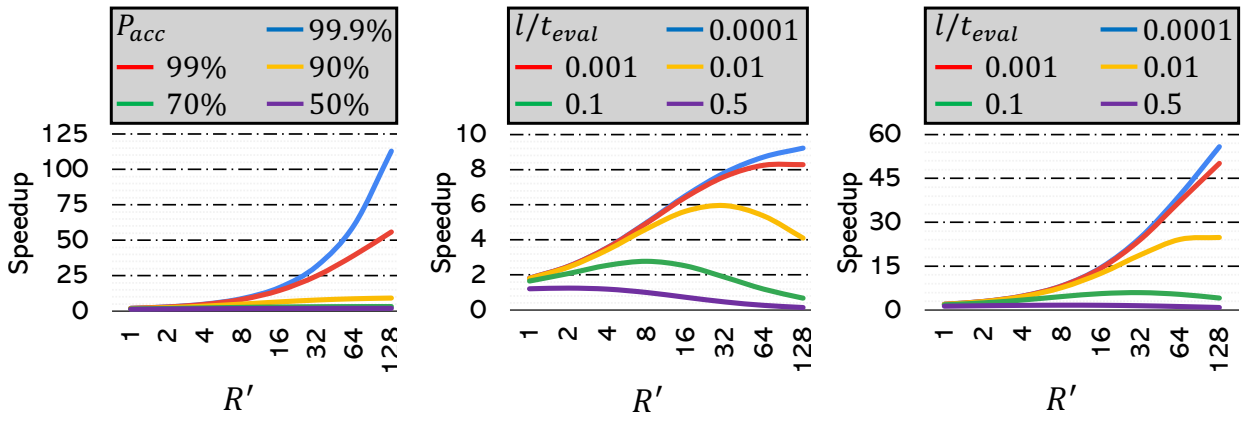
$$P_{cov} \approx \frac{R}{R+n} \times P_{acc} = \frac{R' \times n}{(R'+1)n} \times P_{acc} = \frac{R'}{R'+1} \times P_{acc} \tag{6.6}$$

In slow-expansion applications, we can assume t_{serial}, $t_{threading}$, $t_{load} \ll t_{eval}$. By eliminating these terms and plugging Eq. 6.6 into Eq. 6.5, we can formulate speedup as:

$$Speedup = \frac{T_{Baseline}}{T_{RA^\star}} = \frac{t_{eval}}{(1 - \frac{R'}{R'+1} \times P_{acc}) \times (t_{eval} + R' \times l)} \tag{6.7}$$

[3]The equation does not take the early-termination cases (i.e., when MLS is infeasible) into account.

In this formula, P_{acc} depends on the effectiveness of the employed predictor for the application. R' is dependent on how many threads can be executed in parallel on the system. And, l depends on the predictor design. Figure 6.24 shows how speedup varies with different parameters.



(a) Speedup of different R's with $l = 0.0001 \times t_{eval}$ and varying P_{acc}

(b) Speedup of different R's with $P_{acc} = 90\%$ and varying l

(c) Speedup of different R's with $P_{acc} = 99\%$ and varying l

Figure 6.24: Projected speedup.

The formula suggests that improving RA⋆'s speedup for an application can be achieved in the following ways:

- Improving prediction accuracy: the higher P_{acc}, the higher speedup. Particularly, when the prediction accuracy is high, even the slightest incremental improvement could be of significant importance. For example, in Figure 6.24.a, with $R' = 128$, improving P_{acc} from 90% to 99% results in a 46.6× speedup. In the same configuration, improving the prediction accuracy from 99% to 99.9% brings another 57× speedup. This shows how highly-accurate predictors could greatly accelerate the execution (up to 112.9× in Figure 6.24.a).

- Increasing runahead: by running further ahead, which is viable by using more threads, RA⋆ can further parallelize the algorithm, and further enhance the speedup. However, this projection comes with a caveat: increasing runahead results in increasing the cumulative prediction latency (i.e., $R' \times l$). As such, increasing runahead is good so long as the cumulative prediction latency is outweighed. Therefore, when the prediction latency (i.e., l) is small, increasing runahead yields higher speedups. For example, with $l = 0.0001 \times t_{eval}$, increasing R' from 0 to 128 results in 9.2× speedup when $P_{acc} = 90\%$ (Figure 6.24.b) and 55.9× when $P_{acc} = 99\%$ (Figure 6.24.c).

- Reducing prediction latency: the faster RA⋆ generates the predictions, the better. This is particularly true when R' and P_{acc} are high. For example, in Figure 6.24.c, with $R' = 128$, reducing the prediction latency from $0.001 \times t_{eval}$ to $0.0001 \times t_{eval}$ results in a 5.7× speedup. Also, when the prediction latency is too long ($> 0.1 \times t_{eval}$), RA⋆ not only does not achieve speedup, it causes slowdown, even when the prediction accuracy is high.

This is the behavior we observed with RNNs: although they offer high accuracy, their long prediction latency downgrades their effectiveness. Reducing prediction latency can be achieved by simplifying the prediction method (e.g., *pruning* in the context of deep learning). Likely, it is in conflict with the first approach, i.e., improving prediction accuracy. Typically, there is a trade-off between the performance of a predictor and its overheads; the sophisticated predictors offer a high level of accuracy but impose significant overheads (time) for generating a prediction.

6.7 RACOD and RA* and in The Context of Related Work

6.7.1 Hardware Acceleration for Path Planning

In the context of path planning, a few proposals design accelerators for some narrow domains. Particularly, Murray et al. [200] 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 [182] 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 [200] or costs 768 Kb SRAM storage in [182]. 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*.

6.7.2 Parallel Path Search

Prior work on parallelizing path search in general, and A* in particular, can be divided into two groups: methods that need to (conditionally) *re-expand* nodes to guarantee optimality, and methods that do not.

The first group of methods [127, 172] are effective when expansions are fast; i.e., when evaluations are not costly, which is the case in applications like symbolic task planning. In slow-expansion applications, where evaluations are time-consuming, these methods suffer from significant challenges. Specifically, re-expansions in these methods can happen exponentially-many times, which means the already-slow evaluations can repeat exponentially-many times, hurting performance.

The second group of methods [214] do not require re-expansions to guarantee optimality. As such, they can parallelize and reduce the execution time of slow-expansion applications of A*. These methods, on the flip side, impose non-trivial overheads that limit their performance. For example, PA*SE [214] spends significant time serially searching the states to find the parallelizable ones.

While there are differences in how prior proposals parallelize A*, most of them have one thing in common: they parallelize multiple expansions. In contrast, RA* and *RASExp* run ahead of individual expansions and paves the way for accelerating future expansions; it exploits a different source of parallelism.

6.7.3 Speculative Parallelism

Generally, speculation is a hardware- and system-level technique that uses additional or available-but-idle resources to improve performance. For example, *hardware prefetchers* [119] use additional contexts to predict future memory references of the running application and fetch them from slow memory to fast memory (e.g., from DRAM to on-chip caches); this way, when predicted data are requested, the application enjoys the low latency of the fast memory.

In this chapter, we draw insight from such system-level techniques and propose an algorithm-level speculative approach for parallelizing A*. Throughout the execution, when a node is being expanded, we predict future expansions, pre-evaluate their neighbors on separate threads (i.e., in parallel), and memoize the results. This way, when a predicted node is expanded, we use the memoized results rather than performing time-consuming evaluations, saving significant time.

Outside of the context of path planning, several pieces of prior work have proposed to exploit *ordered parallelism* 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) [250] 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 (and by extension, RA*) 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.

6.8 Chapter Summary

In this chapter, we study path planning, a core module in autonomous robots, and propose *RACOD*, 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. *RACOD* accelerates mobile robot planning by up to 41.4×, with less than 0.3% hardware overhead.

We also propose a generalized form of the technique *(ii)* for a broader set of applications. We propose *Runahead* A*, a method that opportunistically parallelizes the search for slow-expansion applications. *Runahead* A* enhances parallelism by predicting future expansions and pre-computing their expensive operations, thereby reducing execution time. We show that RA* is able to reduce the execution time by up to 14.1× using 16 threads, while preserving A* optimality guarantees.

Chapter 7

Tartan: Microarchitecting a Robotic Processor

This chapter presents *Tartan*, a novel CPU architecture specifically designed for robotics applications. Despite the efficacy of application-specific hardware accelerators in enhancing robotics performance, the rapid evolution of robotics presents a significant challenge. The constant emergence of new algorithms and shifting paradigms quickly renders these specialized accelerators obsolete. Moreover, hardware accelerators struggle to adapt beyond their specifically targeted applications. However, the field of robotics encompasses a vast range of applications, spanning from industrial robots to atmospheric robots. It is unlikely that hardware vendors will implement specialized hardware for each of these diverse robot types individually.

Additionally, prior hardware accelerators for robotics do not tackle memory bottlenecks, and hence face increasing limitations due to the widening gap between computation and memory performance [187, 197].

To tackle these challenges, we introduce *Tartan*, a CPU architecture specifically designed for robotics applications. Our approach includes a detailed evaluation of robotic workloads to identify software bottlenecks and discrepancies between the demands of these workloads and the capabilities of existing architectures. From these insights, *Tartan* is engineered to specifically address and alleviate bottlenecks in common robotic kernels, such as pathfinding, by providing targeted architectural enhancements to boost their performance. The emphasis on prevalent bottlenecks ensures that *Tartan* remains adaptable and forward-looking, capable of supporting a wide range of robots and future algorithms that rely on these critical kernels.

A distinctive feature of *Tartan* is its enhancement of the memory subsystem. *Tartan* incorporates architectural modifications to the cache hierarchy to address memory bottlenecks that hinder robotic applications (see §5.4). This focus on memory-bound performance is often overlooked by hardware accelerators that primarily focus on computational acceleration.

Key architectural features of *Tartan* include the following components.

- **Oriented Vectorization:** *Tartan* supports vectorization of non-contiguous memory accesses exhibited in ori-

ented patterns. These patterns are commonly observed in robotic operations such as ray-casting and collision detection. *Tartan* implements an *explicit*, *in-hardware* address generator to efficiently capture these patterns, departing from the implicitly-contiguous or software-based gather address generation methods used in existing processors.

- **Approximate Acceleration:** *Tartan* offers hardware support for the approximate acceleration of error-tolerant applications. This feature is advantageous for various robotic tasks such as pathfinding and scene understanding, which are capable of tolerating a certain level of (one-sided) error. Specifically, *Tartan* introduces a novel computational paradigm termed AXAR: by leveraging guarantees from specific algorithms, it allows for the approximation of certain computations without affecting the final result.

- **Robot-Semantic Prefetching:** *Tartan* aims to leverage semantic features in robotics for its novel hardware data prefetchers. The semantic information utilized by *Tartan* (e.g., sparse versus dense environmental areas) is relevant across a wide range of robotic applications. This method represents the first effort in the robotics field to leverage application semantic information for hardware prefetching.

- **Intra-Application Cache Partitioning:** *Tartan* introduces a cache management scheme to address intra-application contentions in complex robotic scenarios (e.g., pathfinding in unpredictable terrains). It seeks to implement a (soft) partitioning of the cache space among semantic units (e.g., paths) to optimize cache performance. This method is the first to explore cache partitioning for a singularly running application to *enhance its performance*. It also represents the first exploration into partitioning *private* caches.

- **Engineering Optimizations:** *Tartan* incorporates engineering optimizations tailored for robotics, including adjustments to cacheline size and selective caching policies to enhance the performance of robotic workloads.

Our evaluation of *Tartan* using six end-to-end robotic applications from *RoWild* (see §5.3) demonstrates substantial performance enhancements in all tested scenarios. The observed performance gains span from 1.31× to 3.87×.

The rest of this chapter lays the foundation for designing a domain-specific robotics CPU. It begins by outlining the motivation for such an endeavor, followed by a detailed exploration of performance bottlenecks in robotics. Subsequently, the chapter discusses the architectural features of *Tartan*, the evaluation methodology employed–including the simulation framework, benchmark suite, and baseline architecture–and concludes with a thorough analysis of *Tartan* compared to contemporary hardware accelerators and cache prefetchers, underscoring its efficacy.

Publications & Materials

Tartan is published in [115], and its source code is available at: https://github.com/cmu-roboarch/tartan

7.1 The Need for Efficient Robotics CPU

Tartan is a *CPU* architecture tailored for robotics. While DSPs, GPUs, FPGAs, and ASICs are increasingly popular in robotics [3, 19, 55, 100], CPUs continue to be an indispensable element of *every* robot manufactured to date. We believe CPUs will *remain* a critical component in robotics for the foreseeable future, owing to the following attributes.

- **General-Purpose Processing:** Robotics encompasses a broad array of algorithms, each exhibiting varied computational behaviors. While programmable accelerators like DSPs, GPUs, and FPGAs excel in specific computation models (e.g., SIMD), not all robotics tasks align with these models. Conversely, CPUs are designed to manage a wide spectrum of computation types, adeptly accommodating the expanding computational diversity of robotics algorithms.

- **Single-Thread Performance:** Due to their aggressive architecture and high clock frequencies, CPUs deliver superior single-thread performance compared to a GPU's single core or an FPGA's individual logic block. This capability is essential for providing real-time *latency* in robots, not only for hard-to-parallelize algorithms, but also for gather-scatter parallel algorithms where a main thread initiates multiple worker threads and aggregates their results–both these computation models are prevalent in robotics [191]. For instance, as we demonstrated §5.3, *CPUs outperform GPUs* in robotic workloads characterized by high instruction-level but low thread-level parallelism. Along with quick execution of sequential tasks, the CPU's ability for fast context switching renders it crucial for robots requiring real-time decision-making and control.

- **Reliability:** CPUs, as a mature technology, undergo extensive testing at hardware and software levels, resulting in robust error-handling capabilities and well-documented failure modes. The extensive use of ECC memory and parity checks further bolsters CPU fault-tolerance [130, 150]. This reliability is critical for robots operating in inaccessible or demanding settings, like space, where other platforms like FPGAs struggle to offer as high reliability [244]. For instance, Valkyrie, NASA's robot for space exploration, performs all its operations on three Intel Core-i7 CPUs [34], with only error-tolerant sensor interpretation algorithms on an NVIDIA Quadro 1000M GPU [59], and no FPGA [217].

- **Price:** Achieving widespread adoption of robots across various applications hinges on their cost-effectiveness. The inclusion of hardware accelerators like GPUs and FPGAs can substantially elevate production costs; even applications requiring real-time performance may not always justify the heightened price [155]. As such, CPU may be the sole computing platform in some robots, running the entire robotics software without the benefits of GPU/FPGA acceleration, as it is the case in real-world robots like [2, 16, 62, 81, 95, 96].

Due to the extensive use of CPUs in various robotic applications and their pivotal role in robot performance, robotics CPUs have undergone significant evolution over the past decade, becoming markedly more powerful (e.g., increased transistor count, higher clock frequencies, deeper pipelines, larger caches). This evolution is illustrated by microcontroller-based robots using Raspberry Pi, transitioning from a single-core ARM11 CPU in the Raspberry Pi 1 [69] to a 4-core ARM Cortex-A76 CPU in the Raspberry Pi 5 [70]; Qualcomm's robotics platform evolving from a Kryo 385 CPU in the RB3 [68] to a Kryo 585 CPU with double the last-level cache size in the RB5 [67]; NVIDIA robotics boards upgrading from a quad-core ARM Cortex-A57 CPU with 2MB of total cache in the Jetson TX1 [48] to a 12-core Arm Cortex-A78AE CPU with 9MB of total cache capacity in the Jetson AGX Orin [45]; and Intel's NUC, which is extensively utilized in a variety of robots such as [39,53,65], transitioning from a Celeron 847 processor [33] with a maximum clock frequency of 1.1GHz in its first generation [38] to a Core i9-12900 processor [35] with a maximum clock frequency of 5.1GHz in its twelfth generation [37].

Nevertheless, as we show in this paper, substantial potential for enhancement remains even beyond the capabilities of state-of-the-art processors. We thoroughly explore the architectural implications for robotics and suggest architectural improvements to a cutting-edge robotics CPU, aiming to boost performance across a diverse range of robotic applications.

7.2 The Need for Efficient Memory Systems

Robotics are becoming increasingly data-intensive [109,196,233], fueled by the need to process large volumes of data produced by more precise sensors (e.g., high-resolution cameras and LiDARs, highly sensitive force sensors) and the large number of model parameters necessary for robots to function accurately in the wild. This trend underscores the role of memory systems in processors for efficiently managing data delivery to the processors.

Tartan stands out for its enhancements to the memory subsystem, specifically improving cache hierarchy to alleviate memory bottlenecks in robotics, differentiating it from prior work that focused on computational acceleration.

7.3 Performance Study

In this section, we perform a performance analysis to identify bottlenecks within robotic workloads, aiming to target these areas for optimization. We begin by outlining our methodology, then proceed to discuss the results of the study.

7.3.1 Methodology

System:

Our methodology involves initially establishing a baseline processor model, followed by the integration of our proposed architectural enhancements. We model the baseline processor after Intel Core i7-10610U Processor [34], which is integrated into NASA's Valkyrie [56]. The processor features four OoO cores fabricated in 14nm. It includes L1-D, L2, and a shared L3 cache with sizes of 32KB, 256KB, and 8MB, respectively. The latencies for these caches are 4, 14, and 45 clock cycles. The chip includes two DDR4-2666 channels, which offer a bandwidth of up to 45.8GB/s.

Upgrading Baseline:

We apply engineering optimizations to benchmark *Tartan* against the optimal baseline.

1. We upgrade the baseline processor's vector ISA and hardware from AVX2 to the current leading standard, AVX-512.

2. Recognizing that robotic workloads are adversely affected by excessive unnecessary data movements (UDM), we shrink the baseline cacheline size from 64B to 32B. This adjustment leads to a 1.56× reduction in UDM and yields a slight average performance enhancement.

3. Data structures facilitating producer-consumer communications across stages of the robotics software pipeline are allocated to memory regions managed by write-through policies, through manipulation of memory type range registers [30]. This results in a 9%–43% reduction in L3 cache traffic and a 2%–4% improvement in overall performance.

Framework:

We use ZSim [224] to evaluate our proposal. We run all applications until their completion and report execution time for performance analysis.

Workloads:

We evaluate all six end-to-end robots from the *RoWild* suite, as introduced in §5.3, each modeled based on an actual-world robot design. Table 7.1 details the workloads with the algorithms used in their software and the number of threads used in each stage of the perception→ planning→ control pipeline of the robots.

We tune software for the evaluated processor, leading to a slightly different configurations compared to original *RoWild*'s evaluations on ARM Cortex A57 of NVIDIA Nano board (see §5.3). Although threads outnumber the cores,

Table 7.1: Application parameters. **Bold** algorithms are time-dominant.

Robot	Resembling	Major Algorithms	Pipeline Threads
DeliBot	Spot [9]	**MCL** [254], Greedy [216]	8→ 1→ 1
PatrolBot	Pioneer 3-DX [64]	**MobileNet** [252], EKF [240], PP [209]	1→ 1→ 1 ‖ 4†
MoveBot	LoCoBot [103]	**RRT** [179], **CCCD** [245], PID [247]	1→ 8→ 1
HomeBot	Roomba i7+ [79]	**Point-Based Fusion** [228], BT [154]	8→ 1→ 1
FlyBot	AscTec Pelican [6]	LT [199], **WA*** [216], MPC [144]	1→ 4→ 4
CarriBot	Boxbot [10]	POM [151], **A*** [216], DMP [175]	1→ 4→ 1

† Four threads run network inference in parallel with the robot's software pipeline.

empirical findings indicate these settings as optimal. This can be largely attributed to the uneven distribution of work among threads and the benefits of latency hiding [215].

7.3.2 Bottleneck Analysis

We conduct a thorough evaluation of *RoWild*'s robots on our framework to identify bottlenecks and mismatches between the demands of the workloads and the capabilities of the architecture. The insights gained from this analysis are utilized in designing *Tartan*. Figure 7.1 summarizes the results.

Baseline presents the execution time breakdown for the upgraded baseline processor, while Tartan illustrates how *Tartan*, employing the techniques explained later, focuses on and accelerates bottleneck operations. Below, we detail the execution statistics of applications on the baseline processor. The results and conclusions closely resemble those presented in §5.3, with minor variations in numerical data attributable to differences in the architectures and the use of simulation versus real processor frameworks.

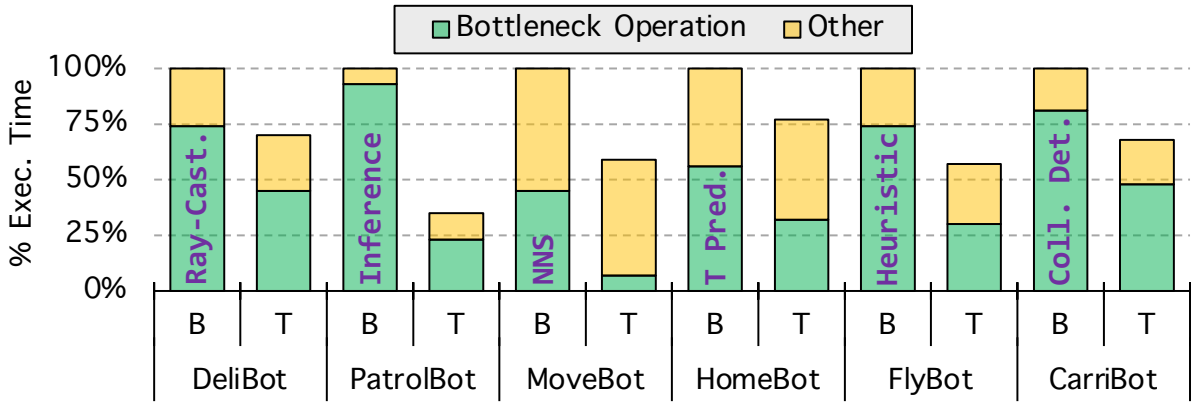


Figure 7.1: Execution time breakdown and bottleneck analysis.

DeliBot utilizes MCL for localization [254], heavily relying on "ray-casting" operations which consume 74% of the end-to-end time. Ray-casting matches laser data with the robot's location hypotheses by checking occupancy in the environment map cell-by-cell. Despite proximity of memory checks, vectorization is unfeasible on current processors

due to misalignment with vector loads.

PatrolBot conducts object detection by feeding captured images into a pre-trained neural network [252] to identify suspicious objects. The neural network inference accounts for 93% of the total processing time.

MoveBot is tasked with moving its arm from one point to another, employing RRT for planning [179]. It uses cuboid-cuboid collision detection (CCCD) [245] to bound obstacles and the robot's body with cubes, checking for intersections during movement planning. CCCD prioritizes speed over accuracy and is parallelized across eight threads, each responsible for assessing collision possibilities with certain obstacles. Without parallelization, CCCD emerges as the primary bottleneck; yet, once parallelized, the major bottleneck shifts to nearest-neighbor search (NNS) operations required by RRT, consuming 45% of execution time. NNS results in irregular memory accesses, challenging existing cache and prefetch techniques in the architecture.

HomeBot uses point-based fusion for 3D reconstruction [228], with 56% of execution time spent on transformation matrix (T) prediction to track the robot's movements. This involves matching point clouds and solving a large linear equation system, including many NNS operations. The irregular memory references from point cloud matching and the heavy floating-point computations for the equations challenge the architecture, stressing memory and processing capabilities.

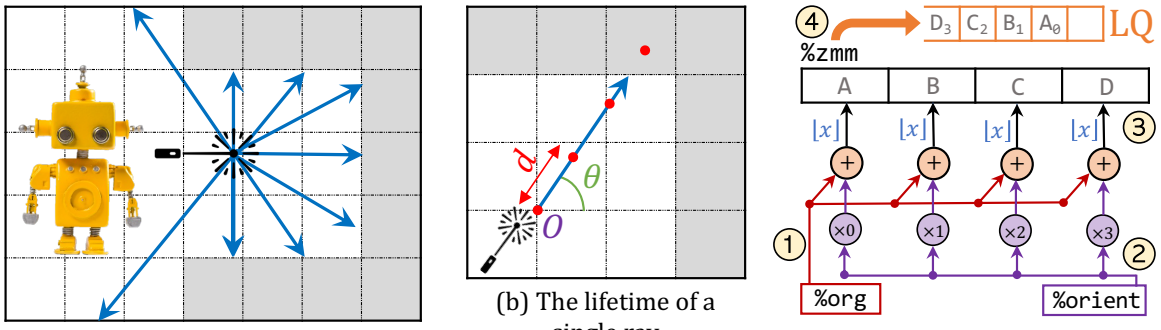
FlyBot conducts aerial photography, requiring frequent relocations in a 3D space. It uses the WA* algorithm for path planning [216], utilizing a sophisticated heuristic function to significantly narrow down its search scope. Yet, the computation of this heuristic function dominates the process, consuming over 74% of the execution time. Additional details about the heuristic function are discussed in §7.5.6.

CarriBot transports sensitive materials within a factory, employing the A* algorithm [216] with precise collision detection in (x, y, θ) space. This collision detection process is time-intensive, consuming over 81% of the execution time. Similar to ray-casting, it requires verifying the occupancy status of various cells in the environment map.

7.4 Oriented Vector Loads

As observed in §5.4.1, data accesses during intensive robotic kernels such as collision detection (§4.2.4) and ray-casting (§4.2.1) manifest in *oriented* patterns. Figure 7.2.a illustrates this using ray-casting as an example. A robot's laser casts rays in different directions to gauge the distance to the nearest obstacle in each direction. Ray-casting is the process of integrating laser-generated distance readings with the robot's pre-existing knowledge (i.e., stored state). During ray-casting, the algorithm scans the map along trajectories that align with the orientation of each emitted ray.

Figure 7.2.b shows the process of ray-casting for a single ray. Let O be the origin of the ray, which is the location of the laser in the environment; let θ be the orientation of the ray with respect to the x-axis; and let d be the step length. The algorithm starts at the origin and iteratively extends the ray's length until it encounters the first obstacle. At step



(a) Ray-casting example

(b) The lifetime of a single ray

(c) Address generation circuit

Figure 7.2: Tartan's oriented vectorization.

i, the algorithm evaluates the location $(O_x, O_y) + i \cdot (d_x, d_y)$, where $dx = d\cos\theta$ and $dy = d\sin\theta$. All the (x,y) pairs generated during ray-casting are floating-point numbers; these are rounded to integers for mapping to the addressable memory grid. For instance, the point $(4.6, 8.5)$ in a 16×16 environment would be flattened to $4.6 \times 16 + 8.5 = 82.1$ and mapped to `env[82]` in memory.

7.4.1 The Problem

In Figure 7.2.b, the red dots denote the points checked during a single ray-casting operation, with each check determining whether a location is free or occupied. Current CPU vectorization approaches, including post-AVX2 gather-scatter instructions (§7.8.1), fail to vectorize such operations: *despite the memory locations checked being nearby and running the same check, the operation cannot be vectorized.*

Consequently, the software needs to sequentially traverse the map on a cell-by-cell basis to perform these checks, leading to excessive processing time. For example, as we reported in §5.3, more than 80% of the end-to-end execution time in two of the six modeled robots are attributed to ray-casting or collision detection–kernels dominated with oriented memory accesses.

7.4.2 Vectorization of Oriented Loads

To address the issue, we propose *Oriented Vectorization (OVEC)*. We incorporate an *extra operand* into the vector load instruction: a register containing the traversal orientation. *OVEC* extends the ISA with the following instruction:

```
O_MOVE %zmm, (%org), %orient
```

`zmm` is the destination vector register for data loading. `org` is the memory source operand, which is a register holding the address of the starting point. In ray-casting, `org` initially holds the origin of the ray. These two operands are similar to conventional vector loads. `orient` is the new operand that *OVEC* introduces, which is a *scalar* register encapsulating the traversal orientation; the flattened representation of (d_x, d_y) in 2D or (d_x, d_y, d_z) in 3D in number

of bytes. For example, with an $N \times N$ occupancy grid, in which every cell stores the occupation probability of an environment location in a `float`, `orient` is $(d_y \times N + d_x) \times$ `sizeof(float)`.

Finally, as in other x86 instructions, the data type is specified in the instruction opcode; e.g., `O_MOVEAPS` and `O_MOVEAPD` for single- and double-precision floating-point, respectively.

7.4.3 Implementation Details

Upon execution of the instruction, the needed addresses must be generated and sent to the memory system. Conventional vector load instructions send only the address of the vector's initial lane, stored in `org`. Addresses of the following lanes are *implicitly* sequential (e.g., `org+1`, `org+2`, ...). However, in oriented vector loads, the addresses for each lane within the vector require *explicit generation*.

Figure 7.2.c shows the address generation process for a vector comprising four lanes. For each lane i within the vector, the address is computed by adding ① the origin address to ② the product of i and the orientation. ③ The fractional parts of the resulting addresses are omitted, and ④ the integral addresses are enqueued into the load queue (LQ). *Parallel address generation via this hardware circuit*, as opposed to serial checking in software, substantially accelerates operations like ray-casting and collision detection, as we show in §7.8.1.

A challenge in oriented vectorization is the accurate data alignment *within* a vector register. The addresses from an oriented vector load correspond to *different lanes* in the register. Thus, when data are fetched from the memory hierarchy, its designated lane within the vector register is unknown.

To tackle this issue, we adopt Intel's approach for *gather* operations: storing the designated lane for loads within LQ (subscripts in Figure 7.2.c). Consequently, when data is arrived, it uses the number in LQ to position itself in the intended lane.

Once data are fully loaded into the vector register, it engages the vector ALU similar to conventional vector instructions, implementing operations as directed by the software. The vector ALU and register file remain unchanged.

7.4.4 Related Work

Prior work on augmenting vector unit capabilities encompasses speculative vectorization support for certain operations [211], employing vector units for runahead execution [202, 203], and data streaming [138]. *Tartan* introduces *OVEC*, a novel design markedly distinct from previous initiatives in terms of design, operation, and targeted applications.

7.5 Approximate-Acceleration

Prior work exploits error-tolerance in tasks like speech recognition, proposing *approximate execution* to trade accuracy for performance. Both software and hardware can implement this approach. For example, NVIDIA's TensorRT [60] uses *quantization*, allowing neural networks to switch from `FP32` to `INT8` representations, which results in faster operations and reduced memory, albeit with some accuracy loss. Similarly, EDEN [173] lowers voltage for DRAM partitions hosting neural network data, sacrificing some accuracy to save energy.

7.5.1 Approximate Execution Accurate Results

In this work, we introduce a new paradigm: *Approximate Execution Accurate Results (AXAR)*. We find that certain computations in robotics can be approximated without altering the *final outcome*, thanks to algorithmic guarantees. We explain and implement *AXAR* in the context of robot path planning.

Path planning refers to the process of finding an efficient (e.g., short) path from a start to a goal point; a key operation in every autonomous robot. As discussed in §4.2.4 and §5.3.5, A*, along with its derivatives [158, 183, 216, 256], is extensively utilized in robotics and other fields for path planning. The intricacies of A* are comprehensively discussed in §6.1.2. Here, we provide only a concise overview of its features relevant to *AXAR*.

Central to A* is its heuristic cost function, which makes it an *informed* search algorithm. For a given state S, the heuristic function, $h(S)$, *estimates* the cost to reach the goal from S. In practice, $h(S)$ can be for example the aerial or Manhattan distance of S to the goal. Using the heuristic function drastically narrows the search compared to uninformed algorithms (e.g., Dijkstra).

A* (with re-expansions permitted) with any *admissible* heuristic outputs the optimal path. An admissible heuristic function h satisfies $h(S) \leq h^\star(S)$ for all S, where $h^\star(S)$ denotes the optimal cost to the goal. This means that *an admissible heuristic never overestimates the cost*. For instance, a heuristic $h(S) = 0$ for all S is technically admissible, but not effective as it provides no insight into the actual cost. Ideally, $h(S)$ should be close to $h^\star(S)$—the closer $h(S)$ is to $h^\star(S)$, the more effectively A* narrows the search.

Calculating the heuristic can be costly in various robotics applications and others (see §6.6). For instance, it may require solving an optimization problem at each step. We propose *AXAR*-acceleration of such cases by approximating the heuristic cost calculation. We argue that *as long as the heuristic admissibility is maintained, the planning outcome will be unaffected* (§7.8.2).

7.5.2 Traditional Approximation

Besides *AXAR*, robotics permits *Traditional Approximation (TRAP)*; trading off some accuracy for performance. For example, while a vacuum robot ideally covers every inch of a floor, occasionally missing a spot does not significantly

affect the overall cleaning outcome. This means there is some leniency in the execution of its scene understanding algorithm.

7.5.3 Hardware-Accelerated Neural Approximation

Both approximation schemes are important. We find that as much as 74% and 56% of the execution time in our workload suite could potentially benefit from *AXAR* and *TRAP*, respectively (§7.8.2). This underscores the importance for the processor to be compatible with both schemes.

To support both *AXAR* and *TRAP*, *Tartan* includes a *Neural Processing Unit (NPU)* [143] tightly coupled to its pipeline. *NPU* is a spatial array of processing elements (PEs), each with a multiply-accumulate (MAC) unit, a lookup table implementing sigmoid-activation, and dedicated buffers for inputs, weights, and outputs, as shown in Figure 7.3.

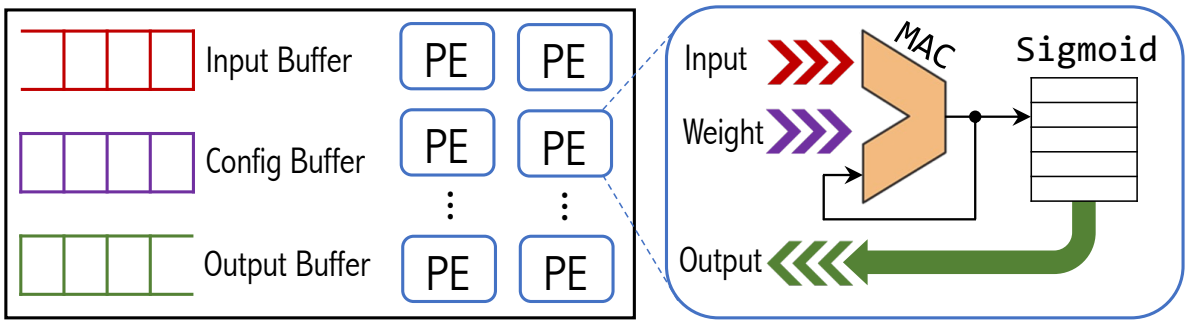


Figure 7.3: Tartan's neural processing unit.

For each function that is costly and targeted for approximation, a neural model is developed, emphasizing *efficiency*. This efficiency is characterized by the model's ability to accurately replicate the outputs of the original function while ensuring the computational footprint does not significantly exceed that of the original. At compile-time, the original function is substituted with this model, facilitating efficient execution by the *NPU* at runtime. Thanks to the *NPU*'s *highly-parallel architecture*, such an efficient neural network can outpace the original, more intensive function when executed on the CPU.

The *NPU*'s configuration, including the number of PEs, can be set based on the available area in the host processor. In §7.8.2, we evaluate various *NPU* configurations.

7.5.4 Why "Neural" Approximation?

The inclusion of the *NPU* and the use of neural models, rather than table-based alternatives, serves a dual purpose: *(i)* Versatility: neural models can learn a broad range of functions, surpassing table-based methods in complex applications [189]. *(ii)* Multimodal: as robotics are increasingly reliant on neural network algorithms, the *NPU* can expedite

both "native" neural networks (i.e., neural models originally employed by the robot software) and "imported" neural networks generated for approximate computing.

Notice *NPU*'s primary purpose is *not* to expedite native neural models; this is rather a secondary benefit. This advantage becomes apparent in scenarios where a robot lacks a GPU or a specific accelerator for neural models, and the *NPU* is available, not being used for approximate acceleration. Thus, it becomes suitable to offload native models to *NPU*. Essentially, for applications that demand large native neural networks, such as advanced perception in autonomous vehicles, the use of GPUs or dedicated accelerators [253] is often necessary, and is complementary to our use of *NPU*.

7.5.5 Software Workflow

Model training is exclusively offline, utilizing multilayer perceptrons (MLPs) for their balance of performance and cost-efficiency in robotics [241]. Network topology and parameters are tailored to each application, considering the acceptable quality loss. §7.8.2 outlines the training data and parameters for each application, the selected network topology, and an analysis of quality loss.

7.5.6 Training for AXAR

In *AXAR*, the inaccuracy of neural models necessitates a *supervisor* to ensure outputs align with algorithmic requirements, similar to the quality controllers in [189, 222]. Unlike these methods that require hardware changes and complex software-hardware co-design, our approach implements supervision in software, integrated *within the algorithmic steps*. Differing from previous methods' emphasis on *predicting erroneous invocations*, our strategy *reduces erroneous predictions* using state-of-the-art training techniques.

We next detail *AXAR* in its application context, applying it to AnyTime A* (ATA*) [183] within FlyBot (see §5.3.5), a drone navigating in 3D. However, it is important to note that the methods and discussions are also relevant to a wide range of other applications, as outlined in §7.5.1.

ATA*

ATA*, widely used in real-time robotics, operates on the principle that *inflating* the heuristic cost with a factor $\varepsilon > 1$ speeds up execution at the expense of generating ε-optimal paths, costing up to ε times the optimal. It starts with a high $\varepsilon = 8$ for a quick initial path, then progressively reduces ε by $step = 1$ to enhance path quality, eventually reaching $\varepsilon = 1$ for the optimal path. Its resilience in unpredictable situations, like delays from unexpected interrupts, makes it popular. In such scenarios, ATA* quickly produces an initial path and continually refines it, or delivers the best path so far, thereby maintaining functionality despite disruptions.

Our method leverages a key aspect of the ATA* algorithm to supervise *AXAR*: each step's path cost does not exceed that of the previous step. The first iteration, with a high ε, runs entirely on the CPU (high ε runs quickly). From the second iteration onward (with lower ε; slower), we offload heuristic cost calculations to the *NPU*. After each iteration's completion (not after every *NPU* invocation), we assess if the current path cost is higher than the previous, indicating *NPU* overestimation. In such cases, the iteration is rerun on the CPU. If not, the process moves to the next iteration. This supervision method introduces minimal overhead, adding a few CPU instructions at the end of each extensive iteration.

This approach ensures that *AXAR* consistently produces outputs within an acceptable range, as the initial iteration runs accurately on the CPU. Thus, *AXAR* maintains the path cost guarantees inherent to the algorithm. Although, in theory, *AXAR* might marginally extend the worst-case execution time by the duration of one NPU-accelerated iteration, the algorithm's design and the fact that the first iteration is CPU-based ensures a key feature: the availability of a viable path even if unexpected events occur post the first iteration. This mirrors the reliability offered by an *AXAR*-less execution.

Training for *AXAR*

FlyBot is a battery-powered drone. It tries to find the shortest path to extend its operation range. It relies on a sophisticated heuristic function to estimate the cost to the goal. The heuristic function calculates the impact of *(i)* aerodynamic drag, *(ii)* altitude change, and *(iii)* wind influence to estimate the cost. Calculating *(ii)* is simple, but computing *(i)* and *(iii)* involves *integrating* over the path, which is computationally expensive.

In our approach, the heuristic function is substituted by a neural model, with an emphasis on *training to minimize CPU rollbacks by minimizing overestimations*. Our training involves an asymmetric, piece-wise loss function which penalizes overestimations more significantly than underestimations:

$$L(y_{\text{true}}, y_{\text{pred}}) = \begin{cases} \alpha \cdot (y_{\text{pred}} - y_{\text{true}})^2 & \text{if } y_{\text{pred}} > y_{\text{true}} \\ (y_{\text{pred}} - y_{\text{true}})^2 & \text{otherwise} \end{cases}$$

Here, $\alpha = 8$ is a constant that determines *how much more* we penalize overestimations. Also, we employ L2 regularization [206] to prevent overfitting by penalizing larger weights. This regularization adds a term $\lambda \sum_i w_i^2$ to the loss function, where $\lambda = 0.01$ is the regularization strength, and w_i denotes the model weights. Lastly, we utilize gradient clipping [193], capping the gradients at $c = 2.5$ during training. This limit is crucial for preventing the model from making excessively large updates, thus aiding in curbing overestimations.

In §7.8.2, we show that, through the employed training techniques, overestimation does not occur during the entire operational period of FlyBot, with more than a million inference operations. Finally, it bears repeating that the entire

process specific to *AXAR* is conducted purely in software. From a hardware standpoint, there is no distinction between *AXAR*, *TRAP*, and native neural network executions.

7.5.7 Related Work

Approximation is a widely explored concept. The innovations of *Tartan* primarily lie in *AXAR*, proposing that certain computations can be effectively approximated based on algorithmic guarantees to yield reliably accurate results. A secondary contribution of *Tartan* involves examining hardware-based approximation for tasks such as T prediction, detailed further in §7.8.2.

7.6 Efficient NNS in High-Dimensional Spaces

Nearest Neighbor Search (NNS) in high-dimensional spaces is essential in key robotic tasks, including motion planning (§4.2.8), scene reconstruction [257], object detection [252], kinematics [210], and sensor data fusion [230].

Popular libraries like OMPL [88] implement NNS using k-d trees and octrees. These data structures, however, present several challenges: *(i)* They lead to inefficient memory access due to scattered node locations, especially in deep octree structures. *(ii)* They do no fully utilize application *semantic* information. Sparse areas result in octrees having many underutilized nodes and k-d trees developing long, data-sparse branches, both leading to inefficient traversal. *(iii)* Octrees are not effective beyond three dimensions, posing a challenge for industrial robots with higher degrees-of-freedom (DoF), which often require searches in higher dimensions (typically 6–7, with cases like the 57-dimensional ASIMO robot [7])

We implement NNS using Locality Sensitive Hashing (LSH), a method for dimensionality reduction. While not the first to use LSH for NNS, our approach is unique in how it capitalizes on the architectural capabilities of modern processors.

7.6.1 Background on LSH

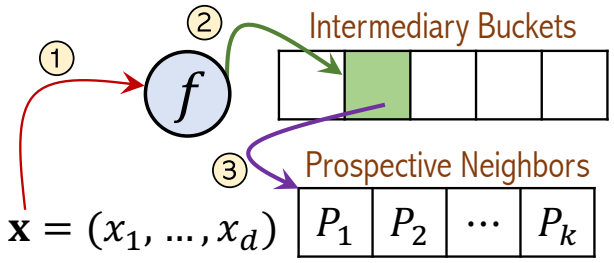
LSH is a technique for reducing dimensionality. It hashes input items so that similar items tend to be mapped to the same "buckets." The fundamental aspect of LSH is its ability to *increase the likelihood of similar items colliding*. In this work, we implement LSH using random projections.

For a point $\mathbf{x} \in \mathbb{R}^d$ (where d is the dimensionality, e.g., 5 for a 5-DoF robot), the hash function is given by $f(\mathbf{x}) = \left\lfloor \frac{\mathbf{x}\cdot\mathbf{r}}{w} \right\rfloor$, where $\mathbf{r}$ is a random d-dimensional vector, each element of which is sampled from a Gaussian distribution $\mathcal{N}(0,1)$, and w controls the bucket sizes in the hash space.

The likelihood of two points $\mathbf{x}$ and $\mathbf{y}$ hashing to the same value is linked to their Euclidean distance—the closer two points are in Euclidean space, the higher the probability they end up in the same bucket.

7.6.2 Approximate NNS using LSH

Figure 7.4.a shows how NNS is performed using LSH. For ① a query point $\mathbf{x}$, the hash function, f, ② assigns it to a specific bucket. This bucket ③ contains a set of points $P_1, P_2, \ldots, P_k$, likely to be near $\mathbf{x}$ in Euclidean space. The algorithm *examines* these points and selects those within a predefined distance threshold ε (i.e., $\|\mathbf{x}-\mathbf{y}\|_2 \leq \varepsilon$) as the nearest neighbors. This selection can include all or a specific number of points, based on the implementation specifics.



(a) Nearest neighbor search using LHS

PC	Region	CD	LD
0x5f	0x4008e1	4	7
0x99	0x40073f	1	1
⋮	⋮	⋮	⋮

(b) Adaptive next-line prefetching

Figure 7.4: Nearest-neighbor search with LSH.

Notably, NNS via LSH is *approximate*, as it relies on LSH's probabilistic properties. Nonetheless, in the context of robotic NNS within high-dimensional spaces, where the utilized algorithms (e.g., RRT, see §4.2.8) inherently accommodate certain levels of error, LSH is an effective approach.

7.6.3 Vectorization of NNS

Vector units are becoming increasingly potent, with AVX-512 featuring 512-bit vector registers. However, the full potential of these units is often underutilized, partly due to limitations in compiler optimization capabilities.

We identify *untapped* potential for vectorization in LSH-based NNS, a domain where existing implementations, including the widely adopted FLANN [21], fall short. We develop a highly-vectorized version of LSH-based NNS, and show that it offers superior performance (§7.8.3). Our approach focuses on vectorizing the projection step (i.e., the dot-product calculation) and aggressively vectorizing the examination process. We call this implementation *Vectorized LSH-Based NNS (VLN)*. Notice, *VLN* is a software approach with no hardware modifications.

7.6.4 Adaptive Next-Line Prefetching

As Figure 7.4.a suggests, access patterns within each bucket are sequential, leading us to employ next-line prefetchers. However, we observed notable variability in the *number* of accesses per bucket, correlating directly with their density. This variance is linked to the *state of the analyzed environment*. For instance, in motion planning, areas densely populated with obstacles result in fewer viable pathfinding points, creating less populated buckets. Conversely, obstacle-free zones yield densely-filled buckets. This trend is observed beyond motion planning, such as in

point cloud manipulation for scene understanding (see §4.2.3), where we noted significant differences in memory access patterns between dense and sparse areas.

To leverage this observation, we introduce an *Adaptive Next-Line Prefetching (ANL)* prefetcher, depicted in Figure 7.4.b. *ANL* operates with two counters per `PC+Region` pair: current degree (CD) and last degree (LD). `PC` is the program counter of load instructions, and `Region` is the high-order bits of load addresses. CD *learns* the prefetching degree (i.e., the number of prefetches issued) for each `PC+Region`, while LD stores the past observations to *issue* prefetches.

ANL employs a 16-entry table, tagged by the concatenation of `PC` and `Region` bits. Upon a cache miss, the table is looked up using the `PC+Region` bits of the load. If the table lookup is a hit, we *(i)* prefetch the number of cachelines indicated by LD, *(ii)* increment CD, and *(iii)* reset LD. If it is a miss, we allocate a new entry, possibly evicting an existing one (see below). New entries start with CD and LD set to 0. When a region is *terminated* (a cacheline of it is evicted from the cache), all *ANL* entries tracking that region *(i)* copy their CD to LD and *(ii)* reset CD. This mechanism enables *ANL* to learn distinct access patterns for each `PC+Region`.

Two details are important to the efficient performance of *ANL*. Firstly, *ANL* needs to use small regions to minimize overprediction. *ANL* bases its prefetching decisions solely on the count of used cachelines. Therefore, in medium-density environments, larger region sizes could lead to significant overprediction. In this work, we utilize 1KB regions. Notice, while the explanation of *ANL* draws parallels between LSH "buckets" and prefetcher "regions", and a correlation exists (different buckets correspond to different memory addresses), it is important to differentiate them: "buckets" are conceptual, at the algorithm level, while "regions" pertain to hardware-level physical address granularities.

Secondly, when *ANL* needs to evict an entry to accommodate a new one, the entry with the lowest `max(CD, LD)` value is chosen for eviction. This approach is hardware-implementable with small tables like *ANL*'s (§7.8.3). The rationale behind this policy is to keep entries with higher degrees, as these are responsible for the majority of prefetch requests. This means that *ANL* is less affected by missing prefetch opportunities in sparser regions, whereas missing such opportunities in denser regions would be more detrimental to its performance.

Finally, notice that *ANL* is not designed merely to accelerate NNS. Rather, it is designed as a *general-purpose* prefetcher for robotics applications, adept at learning and adapting to the *density* of references across different regions *during runtime*. In §7.8, we thoroughly evaluate the effectiveness of *ANL*. Also, *ANL* can prefetch into any cache level; in this work, we put the prefetch requests into the private L2 cache.

7.6.5 Discussion

Our approach effectively addresses the challenges associative with k-d trees and octrees (§7.6). *(i)* Storing points in buckets facilitates cache-friendly, sequential memory accesses. *(ii) ANL* capitalizes on semantic information (e.g., varying densities) within the application. *(iii)* LSH is inherently scalable to higher dimensions, thanks to the dimensionality reduction in its projection phase.

7.6.6 Related Work

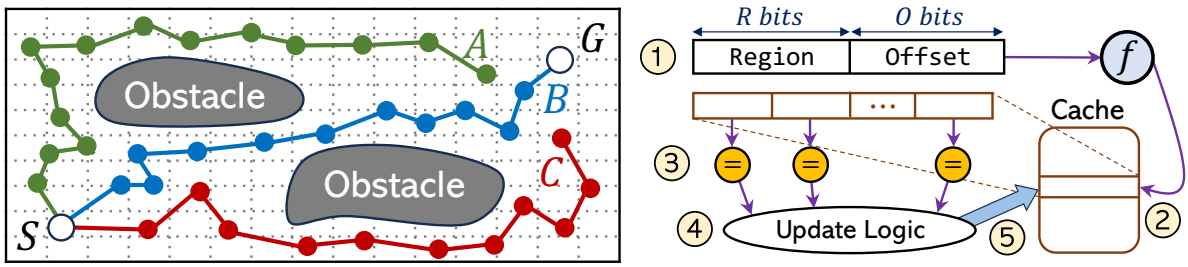
NNS is vital for a broad range of applications and is tackled through a variety of approaches including parallelization [111, 132], compression [140], and application-specific optimizations (e.g., for CNNs [251]). *Tartan* introduces a novel hardware/software strategy that significantly enhances performance. While *Tartan*'s NNS solution can stand alone as a simple and effective method, it can also work orthogonally with existing techniques. For instance, compressing LSH data [140] boosts efficiency without compromising the benefits of *VLN*. Alternatively, *Tartan*'s components, such as *ANL*, can be synergistically combined with other methods where its premise, like data heterogeneity, applies.

7.7 Intra-Application Cache Partitioning

Graph search is crucial in tasks like pathfinding, motion planning, and decision making. While it often exhibits locality, its unpredictability renders the workload memory-intensive [197]. As explained in §6.1.1, in graph search, the robot seeks a *path* from a start to a goal point. This "path" varies by context: in pathfinding, it is the sequence of locations to the goal; in motion planning, it is the configurations for object grasping; and in decision making, it is the set of actions required to perform a task.

Graph algorithms employed across various robotic applications are varied, yet they share a key feature: *concurrent exploration of multiple paths* to determine an efficient, or the most efficient, path. The definition of efficiency varies by application: for drones, it may be the shortest path; for manipulator robots, the smoothest; and for self-driving cars, the path that optimizes fuel efficiency.

Figure 7.5.a illustrates a mobile robot's concurrent exploration of multiple paths in pathfinding, moving from start point *S* to goal *G*. Due to two large obstacles, the route *forks* into three paths: *A*, *B*, and *C*. The algorithm concurrently expands these paths, ultimately choosing one as the final route. This illustration omits two key aspects for clarity: *(i)* the actual number of paths can exceed three, depending on the number and proximity of obstacles; *(ii)* each graph node typically involves extensive neighbor explorations, implying numerous memory accesses to adjacent locations.



(a) Pathfinding example (b) Tartan's cache partitioning scheme

Figure 7.5: Tartan's FCP with an example application.

7.7.1 The Problem

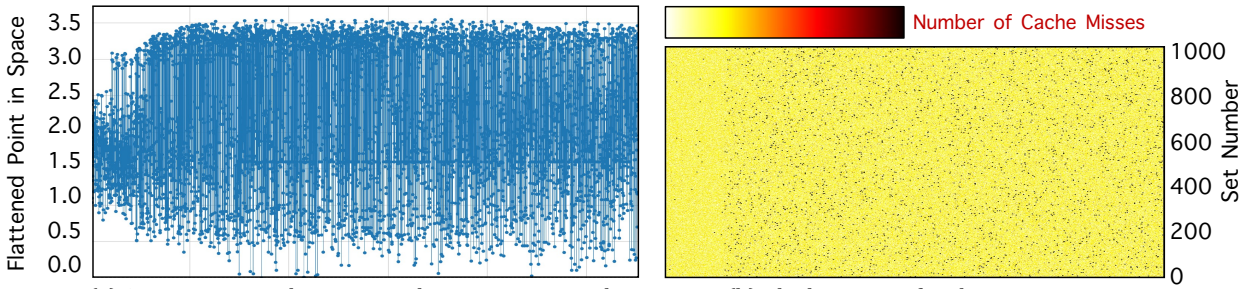
When multiple paths are explored concurrently, they compete for resources, notably hardware caches. This competition leads to paths *evicting each other's data* from the cache. This issue arises even in private caches and is particularly prevalent in single-threaded implementations of widely-used algorithms like A* and RRT, which are challenging to parallelize. The frequent eviction of data from caches negatively impacts the hit ratio, degrading performance.

Importantly, the concurrent exploration of multiple paths is *not* a rare event in many applications. In the A* algorithm and its variants, each iteration involves selecting the node with the highest potential for the optimal path, leading to frequent time-wise switches between paths in $A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow C \rightarrow \cdots$ order. Therefore, each time $C \rightarrow A$ occurs, there is a possibility that some of A's data has been evicted from the cache, resulting in a slowdown. Similarly, in the RRT algorithm, path choices are based on random sampling, which leads to random switches between paths and the same caching challenge.

Figure 7.6 showcases this issue in motion planning with MoveBot (§5.3.3), a 5-DoF robotic arm picking and placing objects in a cluttered environment. It uses the RRT algorithm (see §4.2.8) with 100K random samples (≈1.9MB data working set). Figure 7.6.a depicts the sequence of visited 3D points (flattened), highlighting the algorithm's tendency to oscillate between different environmental areas, repeatedly visiting nodes from various potential paths. Figure 7.6.b presents the L2 cache sets' heatmap, with black dots signifying sets with frequent misses.

This heatmap reveals that a small number of sets (5–15) accounts for most misses. In other words, drawing a random vertical line at any point in steady-state execution will likely intersect with about 5–15 nodes. (The exact distribution of 3D points and hotspot sets are immaterial; the essential insight intended by the figures is the *oscillation* between different areas and the *existence* of hotspot sets.)

Comparing this heatmap with the explored points in Figure 7.6.a reveals a correlation between the number of hotspot sets and *distinct* exploration areas (i.e., far from each other), indicating that multiple actively-explored paths often map to a small number of cache sets, resulting in competition for these sets.



(a) 3D points visited in motion planning in time order (b) The heatmap of cache sets misses

Figure 7.6: Cache sets miss density with motion planning progress.

7.7.2 Cache Partitioning

Partitioning the cache space among different paths can alleviate this issue. However, current cache partitioning solutions like Intel's CAT [43] are unsuitable for several reasons. Firstly, they focus on partitioning *across cores*, which is not applicable to single-thread executions. Secondly, they partition cache by physical ways, an approach not feasible when dealing with potentially tens of paths ($>$ number of ways). Thirdly, these methods often compromise performance to ensure fairness or quality of service, which holds no value in this context. Research proposals in this area [141, 223] similarly encounter one or more of these issues.

In this work, we propose *Fuzzy Intra-Application Cache Partitioning (FCP)*.[1] The key idea of *FCP* is the partitioning of cache by *manipulating replacement metadata*. Specifically, it prioritizes evicting cachelines associated with paths that have excessively used cache capacity. Below, we detail the operations of *FCP*. Unlike CAT [43], *FCP* does not enforce strict cache partitioning but instead implements a "fuzzy" partitioning approach, functioning on a best-effort basis.

FCP is predicated on the understanding that *inter-path* cache contention becomes problematic when paths diverge significantly, each exploring a *distinct, distant memory region*. In other words, when paths traverse spatially close regions, inter-path cache contention is not only unproblematic but also beneficial for spatial locality. For instance, in Figure 7.5.a, the $A \rightarrow B \rightarrow C \rightarrow A$ traversals adversely affect A if both B and C access memory regions far from A. Conversely, if either is spatially proximate to A, the inter-path contention from temporally-interleaved $B \rightarrow C \rightarrow A$ traversals can be advantageous. This is because it might (inadvertently) prefetch data for A if that data resides in the cachelines of B or C.

To address the issue, *FCP* first aims to *map some of the data from individual regions to the same cache sets*. It achieves this by altering the cache's indexing scheme. Considering regions of 2^O cacheline size, the incoming addresses (byte offset excluded) comprise R bits for the region and O bits for the offset within the region, as shown in ① in Figure 7.5.b. With 2^S sets, the standard practice without *FCP* is to use the lower S bits for indexing. When $O < S$, which is almost always the case, *cachelines from a region never map to the same set*. *FCP* seeks to modify

[1] Not to be confused with [198], which involves cache partitioning *across threads*.

this, increasing the likelihood that *some* cachelines within a region map to the same set.

A naive solution could be to index the cache solely with the R bits of the region, causing cachelines of a region to map to the same set. However, this method is detrimental to regions with good spatial locality. The cache's limited associativity prevents storing all or most cachelines from such regions simultaneously, thus failing to exploit the spatial locality.

To strike a balance between locality and partitioning, our approach involves XORing the low-order l bits of the region with the high-order l bits of the offset when ② indexing the cache. This technique introduces some uniform entropy into the indexing process, aiding in achieving a balance between spatial locality and the objectives of *FCP*. Additionally, to ensure compatibility with *Tartan*'s *ANL* prefetcher (§7.6), we exclude the low-order bits of the offset from the XOR operation to prevent cache hotspots induced by the prefetcher. In Section §7.8.4, we will present an experimental analysis to determine the optimal values for l and the region size, thereby choosing the most effective indexing scheme. It is important to note that a bit-wise XOR with one input known is one-to-one, meaning this indexing method does not alter the number of tag bits.

The second component of *FCP* involves manipulating replacement metadata. When a cache fill occurs, either due to a demand miss or prefetching of cacheline X, *concurrently* with tag-checking, ③ cachelines that share the same region bits with X are identified for manipulation. These selected cachelines are then processed through an ④ update logic, which modifies their LRU recency. The update unit executes the function $m(x)$ on each recency number, followed by ⑤ writing the updated metadata back into the cache.

It is important to note that these operations occur concurrently with the cache's baseline functions and are fully-implementable in hardware with minor modifications to existing circuitry. However, the function $m(x)$, which manipulates the recency counters, needs additional circuitry. It alters x to *expedite its eviction from the set*, thus preventing the region from occupying excessive cache capacity.

Applying the $m(x)$ function to recency counters within a set modifies the eviction probability of block x_i to $P_{evict}(x_i) \approx 1 - F(m(x_i))$, where $F(m(x_i))$ represents the cumulative distribution function of the transformed recency counters. (We assume the baseline replacement policy priorities the eviction of lines with *larger* recency counters.) As such, given the non-uniform access patterns typical in graph processing [197], a non-linear function (e.g., quadratic), can enhance performance by more distinctly differentiating the eviction priorities of frequently versus infrequently accessed blocks. We explore various manipulation functions and their effectiveness in §7.8.4. Finally, *FCP* is adaptable to any cache level; for this work, we focus on its implementation in the private L2 cache.

7.7.3 Related Work

To our knowledge, *FCP* is the first effort to partition cache capacity within *one* application to *enhance* its performance. Previous studies on hardware cache partitioning [141, 223, 238, 242] have focused on partitioning cache space among *multiple* applications, aiming to boost aspects such as fairness or quality of service, but often *degrade* the performance.

7.8 Evaluation

We conduct a comprehensive evaluation of *Tartan* using the methodology explained in §7.3.1. We evaluate each component of *Tartan* individually, followed by an assessment of their combination.

7.8.1 Tartan Accelerates Ray-Casting and Collision Detection

Figure 7.7 shows the execution time (bars) and dynamic instruction count (dots) across different methods, normalized to the Baseline processor (with engineering optimizations). We evaluate OVEC, Gather, and RACOD.

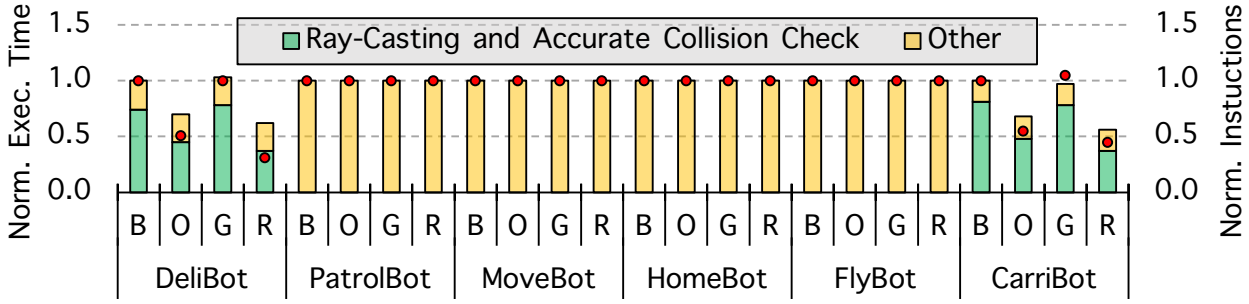


Figure 7.7: Oriented access patterns and different vectorization methods.

OVEC, a component of *Tartan* (§7.4), is estimated to have a latency of 5 cycles for address generation, based on numbers published in [110, 236]. This estimation accounts for the latency of one floating point addition and multiplication, with the latter simplified in hardware due to *(i)* one constant integer input and *(ii)* no need for the output's fractional part.

Gather, a software implementation of *OVEC* using Intel's `VGATHERDPS` [24], serves as a reference. It involves calculating $\lfloor i \times orient \rfloor$ for each lane i (§7.4.3) and arranging them in a vector register *in software*, with `VGATHERDPS` fetching data based on this index vector [30].

RACOD, as introduced in Chapter 6, designed for collision detection in mobile robots, parallelizes address generation in hardware. *RACOD* is not readily applicable to ray-casting. However, to project the speedup that a *RACOD*-like accelerator can achieve for ray-casting, we model a design that performs both address generation and obstacle-checking in hardware.

Results indicate that *OVEC* substantially boosts ray-casting and collision detection, with speedups of 1.64× and 1.69×, respectively. Gather is less effective, as the added instructions for index calculations outweigh the vectorization benefits. *RACOD* outperforms both due to eliminating CPU back-and-fourths; it independently fetches addresses and only interacts with CPU for final outcomes. However, *RACOD*'s superior speedup requires *integrating two separate ASIC units* for ray-casting and collision detection. In contrast, *OVEC* achieves 89%/82% of *RACOD*'s benefits in ray-casting/collision detection, with minimal overheads (§7.8.5).

Finally, it is worth discussing Intel's 10nm ray-casting accelerator [165], which performs in-hardware *trilinear interpolation*, a crucial component of some ray-casting algorithms [213]. This accelerator also includes specialized *local voxel storage* to exploit the locality of nearby 3D voxels during ray-casting. However, it lacks any mechanism for vectorizing memory accesses, a feature central to *OVEC*. Consequently, Intel's accelerator is fully orthogonal to our proposal.

7.8.2 Tartan Significantly Accelerates Approximable Robotics

Table 7.2 details the functions we select for approximate acceleration and their neural network replacements. Note that these functions do not represent the entire spectrum of approximable tasks in robotics. There exist additional tasks (e.g., controlling velocity and acceleration) where exact computation is not strictly necessary. However, for neural acceleration to be beneficial, the tasks must meet two key criteria: *(i)* they should be learnable by an *efficient* neural network, and *(ii)* they must be computationally intensive enough to justify the CPU-*NPU* communication overheads (see below).

Table 7.2: The neural network workloads evaluated.

Type	Robot	Function	Train Data	Topology	Loss Function	Error
AXAR	FlyBot	Heuristic Cost	Freiburg [23]	6/16/16/1	Custom (§7.5.6)	0%
TRAP	HomeBot	T Prediction	ICL-NUIM [157]	192/32/32/6	MSE [176]	6.8%
Native	PatrolBot	Classification	COCO [13]	50/1024/512/1	BCE [176]	1.3%

As discussed in §7.5.6, for FlyBot, we replace the costly heuristic function with a neural model. The model features a topology of 6/16/16/1, which means the network takes 6 inputs (x, y, z coordinates of start and goal), produces 1 output (estimated cost), and has two hidden layers with 16 neurons each. For training, we use a portion of the Freiburg map, distinct from FlyBot's operational area (see §5.3.5), and measure the error by the increased size of the final path.

HomeBot employs a neural model for predicting optimal transformations, contrasting the baseline's ICP algorithm used for minimizing point cloud distances (see §4.2.3). The model's training and test data are entirely separate. The error is the geometric mean of rotation and translation errors.

PatrolBot's object detection uses a convolutional neural network (CNN), but for the *NPU*, a multi-layer perceptron (MLP) accelerator is used. Despite MLP's limitations in image classification due to input data flattening, their broad learning spectrum led to their choice for *NPU*, aiming for a *general-purpose* approximate-accelerator for various robotic tasks. To showcase *NPU*'s effectiveness, PatrolBot's object detection task is implemented with an MLP. We employ principal component analysis (PCA) [170] with $k = 50$ components for dimensionality reduction, training the model with the same dataset as the original CNN. This MLP model on *NPU* proves to be sufficiently accurate and offers reduced execution time compared to the original CNN running on the CPU.

Figure 7.8 compares execution times and dynamic instruction counts across methods, normalized to the Baseline processor. We evaluate Hardware-accelerated and Software-executed neural models, with the former running on a 4-PE *NPU* and the latter implemented using [93] on the baseline processor. We assume a CPU-*NPU* communication latency of 4 clock cycles and 8 clock cycles for MAC operations.

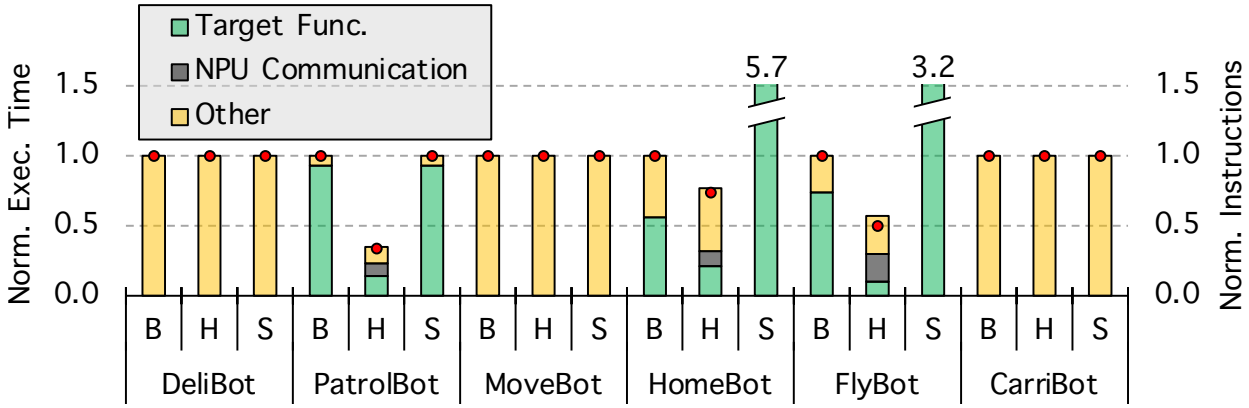


Figure 7.8: Neural acceleration of robotics.

The results indicate substantial speedups with hardware-accelerated neural executions for FlyBot, HomeBot, and PatrolBot (2.7×, 1.52×, and 3.85×, respectively). These speedups are achieved while maintaining acceptable accuracy, as shown in Table 7.2. On the other hand, software-executed neural models suffer from significant slowdowns due to the CPU's inability to exploit neural networks' parallelism.

Table 7.3 explores how varying the number of PEs affects speedup. More PEs allow for increased parallelism of operations, leading to further speed improvements. Given these outcomes, we select a 4-PE design for the *Tartan*'s

Table 7.3: Different neural processing unit configurations evaluated.

PEs	Memory	GMean Speedup	Area [μm^2]
2	10.5KB	1.25×	920
4	18.8KB	1.58×	1661
8	35.3KB	1.68×	3144

NPU. Although increasing PEs to 8 enhances speedup, the primary benefit accrues to PatrolBot, with minimal gains for other robots (see the discussion in §7.5.4).

A 4-PE *NPU* utilizes 18.8KB of SRAM, with 16.5KB dedicated to PEs and 2.3KB for their interconnect [143]. Most of the per-PE area is allocated for storing weights (2KB) and the Sigmoid LUT (512×32 bits), while a smaller portion is used for input/output buffers (64B). The interconnect comprises a bus scheduler (1.25KB), input/output buffers (1KB), and a configuration FIFO (32B), as illustrated in Figure 7.3. The logic area, required for a 32-bit MAC per PE, occupies an insubstantial part of the silicon area (§7.8.5).

Finally, integrating the *NPU* into every core is not necessary. In this work, we consider its integration into just one core. This approach is similar to heterogeneous core architectures in processors such as ARM Big.LITTLE [5], where cores possess varying capabilities. The *NPU* is incorporated into a select core, with the runtime or programmer directing tasks for *NPU* execution to that specific cores.

7.8.3 Tartan Enables Fast Nearest Neighbor Search

This section evaluates HW/SW techniques for NNS in MoveBot and HomeBot, both heavily reliant on NNS. Figure 7.9 shows execution time and L2 misses for various methods. We evaluate Brute-force, VLN, K-d tree, and FLANN, with methods marked with a '+' use *ANL* in hardware. Results are normalized against brute-force search without *ANL*.

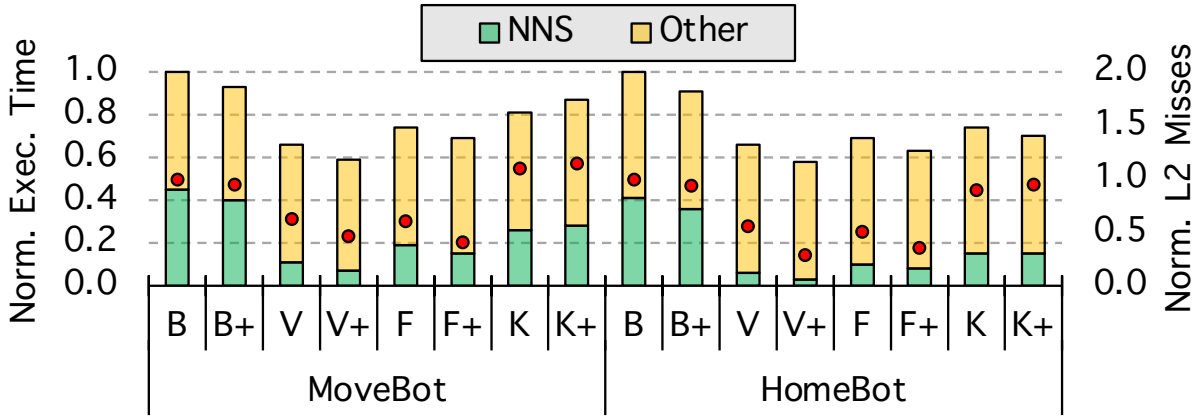


Figure 7.9: NNS with different approaches.

The brute-force search, serving as the baseline, iterates over all points to identify those close to the query point. Our *VLN* employs LSH and vectorization for NNS (§7.6.3). FLANN [21] also uses LSH, but without aggressive vectorization. We tune the bucket sizes (w) for each method to ensure robotic operation accuracy within 1% of the brute-force method (§7.6.1). The k-d tree method uses [49].

Our results show that *VLN*, the software-only technique, not only surpasses brute-force and k-d tree but also significantly outperforms FLANN. The performance gains of *VLN* over brute-force, k-d tree, and FLANN are 5.29×,

2.43×, and 1.7×, respectively. The brute-force approach is exhaustive, searching all nodes, while k-d tree, though an improvement, suffers from costly cache misses. Its misses are often *dependent*, causing full stalls in OoO cores.

The advantage of *VLN* over FLANN stems from its effective use of processor vectorization capabilities. Unfortunately, compilers currently struggle to efficiently vectorize computation patterns like LSH-based NNS, as seen in the FLANN implementation, due to conditional branches in each NNS iteration (§7.6.2). Major compilers, including GCC, Clang, and ICC, fail at this task [108]. Moreover, our results highlight *ANL*'s effectiveness for all methods except k-d tree.

Figure 7.10 evaluates *ANL*. For context, we also examine Next-Line and Bingo [119]. 'Coverage' is the fraction of L2 cache misses covered by the prefetcher, and 'Accuracy' is the fraction of prefetch requests used by the application.

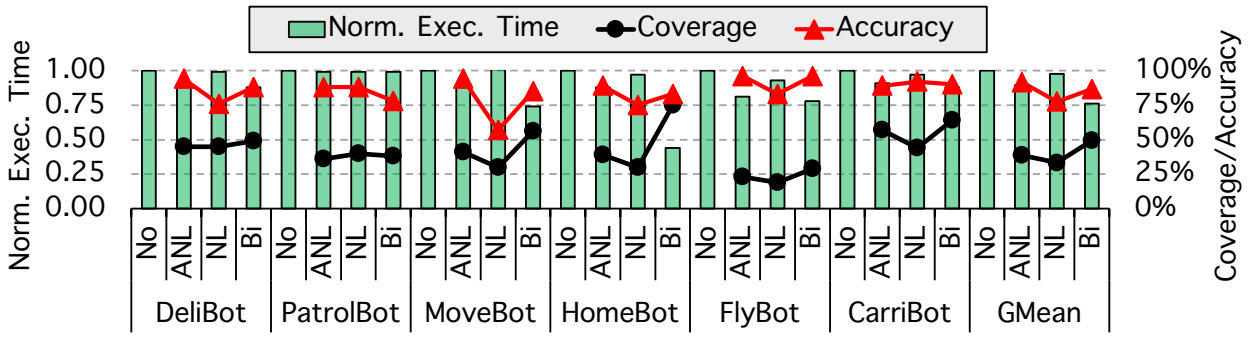


Figure 7.10: Different prefetching approaches.

ANL offers high coverage and accuracy across all workloads, showing its versatility in robotics. It effectively handles the sparse-dense environmental heterogeneity in robots, prefetching memory requests efficiently (§7.6.4). In contrast, Next-Line fails to provide high miss coverage due to the untimeliness of its requests (one prefetch per invoke).

Although Bingo shows higher performance due to its sophisticated pattern learning capabilities, it incurs a significant per-core area overhead of over 100KB for history pattern storage. Conversely, *ANL* matches 85% of Bingo's performance improvement on average with 1000× less area overhead.

In some robots like PatrolBot, prefetchers' high miss coverage does not lead to significant end-to-end speedups. This is typical in compute-bound robots, where the absolute number of cache misses is low, rendering even a high coverage of these misses insufficient for substantial performance gains.

Finally, *ANL*'s metadata table tracks 16 entries, with each entry comprising 12 low-order bits from the program counter plus 38 bits from region addresses for tagging, and 10 bits per entry for recording the current and last degrees. This results in a 120B per core overhead. Also, the logic for implementing the table's replacement policy incurs minimal overhead, requiring only a few integer comparators (§7.8.5).

7.8.4 Tartan Effectively Mitigates Inter-Path Cache Contention

Figure 7.11 assesses *FCP* across various "region size–l bits" configurations and manipulation functions (§7.7.2). The results are normalized to a baseline without *FCP*. The L2 cache is 8-way set-associative in the evaluated processor.

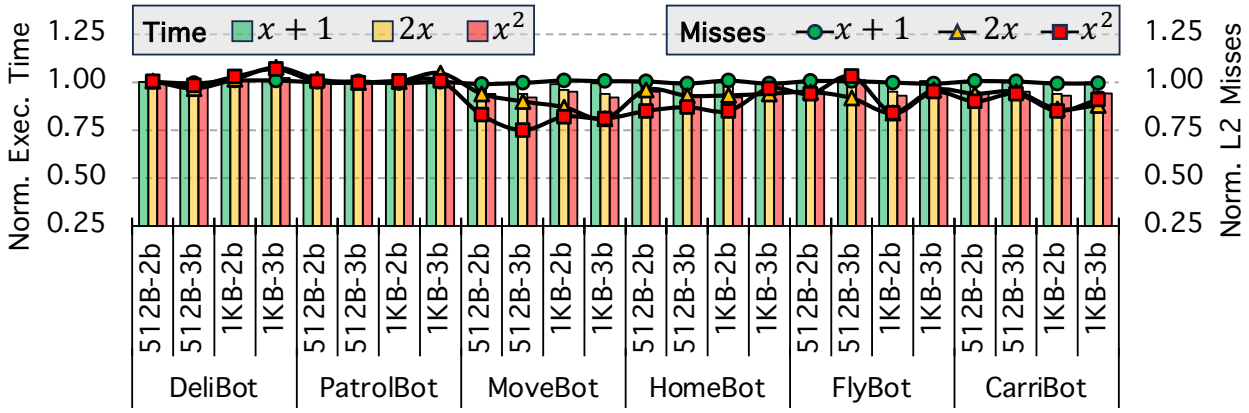


Figure 7.11: FCP with different parameters.

The evaluation shows differing behaviors based on the chosen parameters. For example, $l = 3$ is effective in graph-search–intensive robots like MoveBot but incurs slowdowns in certain scenarios where the underlying assumptions do not apply. We select $l = 2$ bits, setting the region size to 1KB.

Additionally, the results shows the critical role of the manipulation function, $m(x)$, in *FCP*'s efficacy. As expected, $m(x) = x^2$ enhances performance by creating more distinct eviction priorities for cache blocks, as discussed in §7.7.2. The function $m(x) = 2x$ also demonstrates competitive performance, trailing x^2 by only 2.9%. A contributing factor to their similar performance is the three-bit cap on both functions' outputs, aligning with the size of LRU counters in an 8-way cache. We opt for $m(x) = x^2$ in *FCP* due to its superior performance. Notably, the full x^2 logic need not be implemented in hardware; given the known input range (e.g., $\{0, 1, \ldots, 7\}$ for an 8-way cache), a small lookup table can efficiently realize this function.

FCP achieves up to 8% in performance improvement and a 18% reduction in L2 misses. Some applications exhibit only modest gains, mainly because most L2 misses are serviced by the larger 8MB L3 cache, whose latency is somewhat tolerable by the employed aggressive OoO cores. Our analysis indicates that *FCP* is not warranted for L3 in our workloads. However, its general-purpose nature suggests potential for more significant improvements in other scenarios. For instance, in less aggressive robotic CPUs, private cache misses are less tolerable, or in other graph-intensive applications with high L3 miss rates [121], implementing *FCP* for L3 is more justifiable.

Finally, implementing *FCP* with a 16-way L2 cache yields performance improvements akin to those achieved with an 8-way L2. Although graph-search-intensive applications experience a marginally greater improvement and other applications a slightly lesser one, the overall average performance enhancement remains consistent across both configurations.

7.8.5 Putting It Altogether

So far, we evaluated components individually on *modified* software. It is crucial *(i)* to ensure the processor sustains or enhances legacy software performance, and *(ii)* that the components operate in harmonious synergy. Figure 7.12 shows performance of both legacy and *Tartan*-optimized software, each benchmarked against their respective baselines. We evaluate two cases of the *Tartan*-optimized software: *(i) No Approx.* prohibits hardware approximation, requiring exact hardware results and preserving end-to-end accuracy, and *(ii) Approx.* permits hardware approximation, allowing for the substitution of neural models for ICP and MLPs for CNNs.

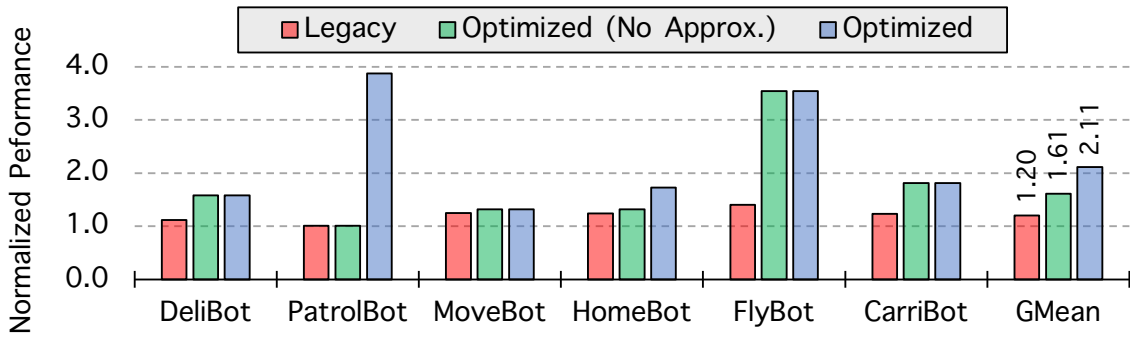


Figure 7.12: Tartan's performance with legacy and optimized software.

The results show that *Tartan* not only maintains the performance of legacy software but also enhances it by 1.2× (up to 1.4×). The hardware-only techniques, *ANL* and *FCP*, contribute to performance improvements for both legacy and optimized software, while *OVEC* and *NPU* improve optimized software, as they need new machine instructions to be utilized. For optimized software without approximation, *Tartan* achieves a speedup of 1.61× (up to 3.54×); when approximation is allowed, the speedup further increases to 2.11× (peaking at 3.87×).

Additionally, the results affirm the seamless integration of *Tartan*'s components, preserving their individual performance benefits. An exception is noted in the combination of *NPU* and *ANL* with the optimized software. Here, the integration of *NPU* leads to substituting the T estimation in the ICP algorithm with a neural approach, thereby removing the NNS operations necessary in the baseline algorithm from the approximate version. This change results in a diminished impact of *ANL*, as the opportunities for speedup are reduced.

Finally, Table 7.4 shows the estimated overheads for each component, using data from [110, 236]. Assuming a mobile die area of 133mm^2 in 14nm [42], the overall overhead for *Tartan* is merely 0.001%.

The overhead in *OVEC* arises from the logic used for address generation. For *NPU*, the overhead is attributed to its PEs and their interconnections. The main source of overhead in *ANL* is its metadata table. Finally, the overhead of *FCP* mainly stems from its 8-entry lookup table per L2 cache, which facilitates the implementation of the manipulation function.

Table 7.4: Overhead breakdown.

Component	**Memory**	**Area [μm^2]**
4 × OVEC	—	258
1 × NPU	18.8KB	1661
4 × ANL	480B	30
4 × FCP	12B	1
Total	19.3KB	1949

7.9 Chapter Summary

Recent studies underscore efficiency shortfalls in contemporary processors when executing robotic workloads. Typically, these processors are optimized for a broad spectrum, from standard desktop applications to burgeoning machine learning tasks, yet they fall short in fully addressing the unique and multifaceted demands of robotics, which include a spectrum of tasks from computer vision to predictive analytics.

This chapter introduced *Tartan*, a CPU architecture specifically designed for robotics. *Tartan* aims to rectify the limitations of current processors by integrating targeted architectural advancements for more efficient robotic task execution. *Tartan* boosts the performance of legacy robotic software by 1.2× (up to 1.4×), non-approximable software optimized for *Tartan* by 1.61× (up to 3.54×), and approximable software optimized for *Tartan* by 2.11× (up to 3.87×).

Chapter 8

Conclusion and Future Directions

The rapid advancement of robotics technology is significantly altering societal dynamics, exerting a profound impact on various aspects of daily life. This transformation spans numerous areas, including industrial automation, healthcare, personal assistance, and environmental monitoring. As robots grow increasingly sophisticated, their integration into diverse sectors is effecting a paradigm shift in the execution of tasks and delivery of services.

To maximize the potential of robotics, it is imperative to bridge the long-standing divide between robotics and computer architecture research. This nexus is crucial for the development of robotic systems that are more efficient, intelligent, and adaptable. Fusing insights from computer architecture with robotics can yield substantial enhancements in processing capabilities, energy efficiency, and decision-making processes in robotic systems.

In this thesis, we set out to bridge this gap by making the following contributions.

- **RTRBench:** A comprehensive, open-source benchmark suite for *individual* robotics tasks. RTRBench encompasses a broad spectrum of tasks covering the entire software pipeline of most autonomous robots. *RTRBench* comprises 16 tasks across robot perception, planning, and control stages. Together with the suite, we conduct an evaluation of the workloads at the architecture level and suggest opportunities for performance improvements. *RTRBench* is published in [116].

- **RoWild:** A comprehensive, open-source robotics benchmark suite coupled with a systematic performance study, focusing on *end-to-end* robotics applications and supporting multi-platform execution. *RoWild* models six different end-to-end robotic applications and evaluates them on various platforms, from low-end embedded CPUs to high-end server-grade GPUs. Additionally, *RoWild* investigates the architectural implications of robotics running on modern hardware, examining aspects like caching, prefetching, and vectorization effectiveness. *RoWild* is published in [113, 114].

- **RACOD:** An algorithm/hardware co-design for mobile robot path planning. *RACOD* exploits architectural-

level computation characteristics of mobile robot path planning, as well as its high-level semantic features, to massively parallelize the task. It architects cheap hardware accelerators to exploit fine-grained parallelism and spatial locality in costly collision detection operations, and it enables proactive exploration by semantically predicting directions along the search path. *RACOD* accelerates mobile robot planning by up to 41.4×, with less than 0.3% hardware overhead. *RACOD* is published in [112].

- **RA⋆:** A speculative parallelism scheme in the software stack of robots to enhance robot planning performance and other applications of the A⋆ algorithm. *RA⋆* enhances parallelism by predicting future expansions and precomputing their expensive operations, thereby reducing execution time. *RA⋆* is able to reduce the execution time by up to 14.1× using 16 threads, while preserving A⋆ optimality guarantees. *RA⋆* is published in [118].

- **Tartan:** A domain-specific processor designed for robotics applications. *Tartan* aims to rectify the limitations of current processors by integrating targeted architectural advancements for more efficient robotic task execution. The architecture effectively addresses both computational and memory bottlenecks, marking a significant advancement over previous works. Key features of *Tartan* encompass architectural support for oriented vectorization, approximate acceleration with accurate outcomes, robot-semantic prefetching, and intra-application cache partitioning. *Tartan* boosts the performance of legacy robotic software by 1.2× (up to 1.4×), non-approximable software optimized for *Tartan* by 1.61× (up to 3.54×), and approximable software optimized for *Tartan* by 2.11× (up to 3.87×). *Tartan* is published in [115].

Following, we explore future directions for research.

8.1 Extending The Benchmark Suite

The field of robotics is in a state of continual evolution, marked by an abundance of research publications and industrial innovations each year. Recognizing the impossibility of encompassing the entire breadth of this rapidly expanding field in a single study or a static benchmark suite, the future work will involve regularly augmenting the introduced benchmark suites. This augmentation will include the addition of new algorithms, applications, and analytical methods to reflect the ongoing progress in the field. Furthermore, we actively welcome and encourage contributions from the open-source community, acknowledging the significant role that collaborative efforts play in enhancing and maintaining the relevance and comprehensiveness of these resources.

8.2 Quantitative Metrics for Evaluating Platforms

Throughout this thesis, we have demonstrated that the computational demands of diverse robotic tasks and applications vary significantly. This variation suggests that certain platforms may be more suitable for specific contexts than

others. Robotics platforms differ in aspects such as cost and Thermal Design Power (TDP), among other key features, necessitating careful consideration to identify the most *efficient computational platform* for each robotic application.

In other domains, existing research has established quantitative criteria for evaluating computational platforms, with metrics such as performance per watt and performance per unit area being widely recognized [188]. However, in the field of robotics, these metrics often do not fully capture the unique requirements. For example, the proportion of power dedicated to computing in robotics, unlike in data centers, constitutes a minor part of the overall power consumption (see §2.4). Similarly, the metric of performance per area might not be entirely indicative of a platform's suitability for robotics. Furthermore, when considering metrics like performance relative to the Total Cost of Ownership (TCO), it is crucial to acknowledge that the cost of a robot is often significantly influenced by non-computational components. For example, a self-driving car valued at over $100,000 may comfortably incorporate processors worth thousands of dollars, in contrast to a budget-restricted wildlife conservation robot.

The challenge of determining the optimal computational platforms for robotics is compounded by the necessity of real-time functionality, which is critical not only for user experience but also for safety. In autonomous vehicles, for example, the rapid detection of obstacles and immediate response is paramount for preventing accidents. Future research in the intersection of robotics and architecture will thus focus on developing quantitative criteria tailored for designing efficient robotic processors. This will include considerations of safety, raw performance, cost per user ownership, potential revenue, and other relevant factors.

8.3 Beyond Performance

In this thesis, the primary emphasis in developing techniques, including the *Tartan* processor, was centered on performance enhancement. However, looking forward, there is a compelling case for broadening the scope of research to encompass other critical aspects that are vital for the holistic advancement of real-time robotic systems. These aspects include, but are not limited to, cyber-security, user privacy, and error resilience.

8.3.1 Cyber-Security

As robotic systems become increasingly integrated into various sectors, their vulnerability to cyber-attacks escalates. Future work could involve developing robust security protocols and architectures within robotic systems to safeguard against unauthorized access and potential cyber threats. This is particularly crucial for applications involving sensitive data or critical operations.

8.3.2 User Privacy

With robots often operating in personal or sensitive environments, ensuring user privacy is paramount. Future research could focus on designing privacy-preserving algorithms and data handling processes. This involves not only securing communication channels but also implementing data anonymization techniques and privacy-by-design principles in robotic systems.

8.3.3 Error Resilience

The reliability of robotic systems in the face of errors or unexpected conditions is a significant concern. Future developments could explore advanced error detection and correction mechanisms, as well as self-healing architectures that allow robotic systems to maintain functionality and safety even when components fail or encounter anomalies.

Bibliography

[1] "3DR Solo Drone," https://uavsystemsinternational.com/products/3dr-solo-drone. 11

[2] "ABB IRB 1200," https://new.abb.com/products/robotics. 109

[3] "An Unprecedented Edge AI and Robotics Platform," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/. 109

[4] "Arduino Ant Hexapod Robot," https://antdroid.grigri.cloud. 69

[5] "ARM Big.LITTLE Architecture," https://www.arm.com/technologies/big-little. 129

[6] "AscTec Pelican," https://www.aeroexpo.online/prod/ascending-technologies/product-181442-24426.html. 4, 43, 47, 111

[7] "ASIMO Specifications," https://asimo.honda.com/asimo-specs. 120

[8] "Boston Dynamics' Atlas," https://www.bostondynamics.com/atlas. 4, 43

[9] "Boston Dynamics' Spot Robot," https://www.bostondynamics.com/products/spot. 4, 43, 47, 111

[10] "Boxbot Launches Last-Mile, Self-Driving Parcel Delivery System," https://www.roboticsbusinessreview.com/supply-chain/boxbot-launches-last-mile-self-driving-parcel-delivery-system. 4, 43, 47, 111

[11] "Carnegie Mellon University, Wean Hall (WEH)," https://www.cmu.edu/computing/services/teach-learn/tes/classrooms/locations/wean.html. 22, 48

[12] "Clearpath Robotics' Grizzly Robot," https://clearpathrobotics.com/blog/tag/grizzly. 4, 43

[13] "COCO Dataset," https://cocodataset.org. 128

[14] "CppRobotics," https://github.com/onlytailei/CppRobotics. 3, 20, 38, 42, 44, 46

[15] "Custom Unreal Environment - AirSim," https://microsoft.github.io/AirSim. 42

[16] "Dual-Arm YuMi - IRB 14000," https://new.abb.com/products/robotics/robots/collaborative-robots/yumi/dual-arm. 109

[17] "EVGA GeForce GTX Titan X Superclocked Graphics Card," https://www.techpowerup.com/gpu-specs/evga-gtx-titan-x-superclocked.b3253. 45

[18] "FACT SHEET: Biden-Harris Administration Announces New Actions to Promote Responsible AI Innovation That Protects Americans' Rights and Safety," https://www.whitehouse.gov/briefing-room/statements-releases/2023/05/04/fact-sheet-biden-harris-administration-announces-new-actions-to-promote-responsible-ai-innovation-that-protects-americans-rights-and-safety. 13

[19] "FANUC Robotics Products," https://www.fanucamerica.com/products/robots/. 109

[20] "FANUC's M-2000iA/2300 Robot," https://www.fanucamerica.com/products/robots/series/m-2000ia/m-2000ia-2300-heavy-payload-robot. 4, 43

[21] "FLANN - Fast Library for Approximate Nearest Neighbors," https://github.com/flann-lib/flann. 121, 130

[22] "Franka Emika's Panda Robot," https://www.pomorobotics.com/robots/frankpanda. 4, 43

[23] "Freiburg Campus 360 Degree 3D Scans," http://ais.informatik.uni-freiburg.de/projects/datasets/fr360. 26, 47, 56, 57, 79, 93, 128

[24] "Gather Packed Single, Packed Double with Signed Dword Indices," https://www.felixcloutier.com/x86/vgatherdps:vgatherdpd. 127

[25] "Global Robotics Market Size, Share, Growth Analysis, By Application (Industrial, and Services), By End-User (Manufacturing, Healthcare) - Industry Forecast 2024–2031," https://www.skyquestt.com/report/robotics-market. 12

[26] "Husky UGV," https://clearpathrobotics.com/assets/guides/foxy/husky/index.html. 4, 43

[27] "Industrial Robotics Market Size, Share & Trends Analysis Report By Application (Handling, Assembling, Processing), By End-Use (Automotive, Electronics), By Region, And Segment Forecasts, 2023–2030," https://www.grandviewresearch.com/industry-analysis/industrial-robotics-market. 12

[28] "Industrial Robots Market," https://www.fortunebusinessinsights.com/industry-reports/industrial-robots-market-100360. 12

[29] "Inside Japan's Long Experiment in Automating Elder Care," https://www.technologyreview.com/2023/01/09/1065135/japan-automating-eldercare-robots. 13

[30] "Intel 64 and IA-32 Architectures Software Developer Manuals," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. 111, 127

[31] "Intel Advisor: Design Code for Efficient Vectorization, Threading, Memory Usage, and Accelerator Offloading," https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html. 44, 61

[32] "Intel Atom Processor Family for Network Applications," https://www.intel.com/content/www/us/en/products/details/processors/atom.html. 17

[33] "Intel Celeron Processor 847," https://ark.intel.com/content/www/us/en/ark/products/56056/intel-celeron-processor-847-2m-cache-1-10-ghz.html. 110

[34] "Intel Core I7-10610U Processor," https://www.intel.com/content/www/us/en/products/sku/201896/intel-core-i710610u-processor-8m-cache-up-to-4-90-ghz/specifications.html. 109, 110

[35] "Intel Core I9-12900 Processor," https://www.intel.com/content/www/us/en/products/sku/134597/intel-core-i912900-processor-30m-cache-up-to-5-10-ghz/specifications.html. 110

[36] "Intel Core Processor Family," https://www.intel.com/content/www/us/en/products/details/processors/core.html. 17

[37] "Intel NUC 12 Extreme / Pro X," https://www.intel.com/content/dam/support/us/en/documents/intel-nuc/NUC12DCM_NUC12EDB_TechProdSpec.pdf. 110

[38] "Intel NUC Board DCP847SKE," https://ark.intel.com/content/www/us/en/ark/products/71620/intel-nuc-board-dcp847ske.html. 110

[39] "Intel NUC10i5 (Fully Configured)," https://roverrobotics.com/products/intel-nuc. 110

[40] "Intel Reserach Lab; The Raw Log Data Was Provided by Dirk Haehnel," http://www2.informatik.uni-freiburg.de/%7Estachnis/datasets/datasets/intel-lab/intel.gfs.png. 47, 58

[41] "Intel Xeon Gold 5218R Processor," https://ark.intel.com/content/www/us/en/ark/products/199342/intel-xeon-gold-5218r-processor-27-5m-cache-2-10-ghz.html. 45, 93

[42] "Intel's Broadwell-U Arrives Aboard 15W, 28W Mobile Processors," https://techreport.com/news/intels-broadwell-u-arrives-aboard-15w-28w-mobile-processors. 133

[43] "Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 V4 Family," https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html. 124, 125

[44] "Jackal Unmanned Ground Vehicle," https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle. 4, 43

[45] "Jetson AGX Orin Series," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin. 63, 110

[46] "Jetson Developer Kits," https://developer.nvidia.com/embedded/jetson-developer-kits. 16

[47] "Jetson Nano Developer Kit," https://developer.nvidia.com/embedded/jetson-nano-developer-kit. 45

[48] "Jetson TX1 Module," https://developer.nvidia.com/embedded/jetson-tx1. 110

[49] "KDTree," https://github.com/crvs/KDTree.git. 130

[50] "KUKA," https://www.kuka.com/en-us. 4, 43

[51] "KUKA KR 60-3 Robot," https://robodk.com/robot/KUKA/KR-60-3. 4, 43

[52] "LBR iiwa," https://www.kuka.com/en-us/products/robotics-systems/industrial-robots/lbr-iiwa. 4, 43

[53] "LoCoBot," https://www.trossenrobotics.com/locobot-base.aspx. 110

[54] "Maximize The Efficiency of Your Logistics Operations with Robots from MiR," https://www.mobile-industrial-robots.com. 4, 43

[55] "Modern Hardware Platforms Used in Robotics," https://evergreen.team/articles/how-to-create-robots.html. 109

[56] "NASA Humanoid Robot to Be Tested As Remote Oil Rig Attendant," https://www.theregister.com/2023/07/10/nasa_to_test_humanoid_robot. 110

[57] "NVIDIA DRIVE End-To-End Solutions for Autonomous Vehicles," https://developer.nvidia.com/drive. 44

[58] "NVIDIA Nsight Systems," https://developer.nvidia.com/nsight-systems. 44, 53

[59] "NVIDIA Quadro 1000M," https://www.techpowerup.com/gpu-specs/quadro-1000m.c1431. 109

[60] "NVIDIA TensorRT," https://developer.nvidia.com/tensorrt. 44, 115

[61] "Phantom 4 Pro - DJI," https://www.dji.com/phantom-4-pro. 4, 43

[62] "PicoGo," https://www.waveshare.com/wiki/PicoGo. 109

[63] "Pin 3.25," https://software.intel.com/sites/landingpage/pintool/docs/98650/README. 64

[64] "Pioneer 3-DX," https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA.pdf. 4, 43, 47, 111

[65] "Powered by Intel Technology, Robot Helps Retail Customers Find the Right Computer," https://www.intel.com/content/www/us/en/newsroom/news/with-intel-tech-robot-assists-retail-customers.html#gs.5h69r1. 110

[66] "PythonRobotics," https://github.com/AtsushiSakai/PythonRobotics. 3, 38, 42, 46

[67] "Qualcomm QRB5165 SoC for IoT," https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/qrb5165-soc-product-brief_87-28730-1-b.pdf. 110

[68] "Qualcomm Robotics RB3 Platform (SDA/SDM845)," https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/robotics-rb3-platform-product-brief.pdf. 110

[69] "Raspberry Pi 1," https://www.pololu.com/product/2760. 110

[70] "Raspberry Pi 5," https://www.raspberrypi.com/products/raspberry-pi-5/. 110

[71] "Raspberry Pi Robot Kit," https://www.raspberrypi.org. 16

[72] "Robotics Automation for Warehousing, 3PLs, Distribution, Manufacturing," https://fetchrobotics.com. 4, 43

[73] "Robotics Industry Size & Share Analysis - Growth Trends & Forecasts (2024–2029)," https://www.mordorintelligence.com/industry-reports/robotics-market. 12

[74] "Robotics Market Forecast: Top Trends That Will Affect Robotics in 2024," https://investingnews.com/robotics-forecast. 12

[75] "Robotics Market Size, Share, Competitive Landscape and Trend Analysis Report by Application and End User: Global Opportunity Analysis and Industry Forecast, 2021–2030," https://www.alliedmarketresearch.com/robotics-market-A13537. 12

[76] "Robotics Research: How Asia, Europe and America Invest – Global Report 2023 by IFR," https://ifr.org/ifr-press-releases/news/robotics-research-how-asia-europe-and-america-invest. 13

[77] "Robots and International Economic Development," https://itif.org/publications/2021/01/25/robots-and-international-economic-development. 12

[78] "Roomba 980 Robot Vacuum," https://www.irobot.com/en_US/roomba-vacuuming-robot-irobot-roomba-restored-980/R980R99.html. 4, 43

[79] "Roomba I7+ Self-Emptying Robot Vacuum," https://www.irobot.com/en_US/roomba-vacuuming/robot-vacuum-irobot-roomba-i7-plus/I755020.html. 4, 43, 47, 54, 111

[80] "ROS - Robot Operating System," https://www.ros.org. 3, 21, 42, 44, 46

[81] "SCARA Robot," http://www.innovativerobotics.com. 109

[82] "Search-Based Planning Lab," http://www.sbpl.net. 20, 46

[83] "Size of the Global Market for Industrial Robots from 2018 to 2020, with a Forecast for 2021 Through 2028," https://www.statista.com/statistics/760190/worldwide-robotics-market-revenue. 1, 12

[84] "Skydio," https://www.skydio.com. 4, 43

[85] "SoftBank Robotics' Pepper Robot," https://us.softbankrobotics.com/pepper. 4, 43

[86] "Streamline On-Board Computing with Our Intelligent Robot Systems," https://www.qualcomm.com/products/internet-of-things/industrial/industrial-automation. 17

[87] "TALOS: The Walking Humanoid Robot That Integrates the Latest Cutting-Edge Robotics Technology," https://pal-robotics.com/robots/talos. 4, 43

[88] "The Open Motion Planning Library," http://ompl.kavrakilab.org. 3, 20, 41, 42, 120

[89] "The Rise of Robots in Defence," https://www.rowse.co.uk/blog/post/the-rise-of-robots-in-defence. 12

[90] "The (Robotic) Doctor Will See You Now," https://www.weforum.org/agenda/2021/03/why-robots-can-be-beneficial-in-healthcare. 12

[91] "The Role of Robotics in Agriculture," https://www.challenge.org/knowledgeitems/the-role-of-robotics-in-agriculture. 12

[92] "TIAGo," https://pal-robotics.com/robots/tiago. 4, 43

[93] "tiny-dnn: Header Only, Dependency-Free Deep Learning Framework in C++14," https://tiny-dnn.readthedocs.io. 128

[94] "TurtleBot," https://www.turtlebot.com. 4, 43

[95] "TurtleBot3," https://emanual.robotis.com/docs/en/platform/turtlebot3/overview. 4, 43, 109

[96] "uArm Swift & uArm Swift Pro Specifications," http://download.ufactory.cc/docs/en/uArm-Swift-Specifications-171012.pdf. 109

[97] "Unity Real-Time Development Platform," https://unity.com. 42

[98] "UR10e," https://www.universal-robots.com/products/ur10-robot. 4, 43

[99] "UR5e," https://www.universal-robots.com/products/ur5-robot. 4, 43

[100] "Yaskawa: Intel FPGA in Robot Controllers," https://www.intel.com/content/www/us/en/customer-spotlight/stories/yaskawa-customer-story.html. 109

[101] "YuMi - IRB 14000," https://new.abb.com/products/robotics/robots/collaborative-robots/yumi/irb-14000-yumi. 4, 43

[102] "Intel Xeon Processor E5-2670," https://ark.intel.com/content/www/us/en/ark/products/64595, 2012. 38, 77

[103] "LoCoBot: An Open Source Low Cost Robot," http://www.locobot.org, 2012. 4, 21, 43, 46, 47, 52, 77, 81, 88, 111

[104] "1st Summer School on Cognitive Robotics," http://cognitive-robotics17.csail.mit.edu, 2017. 33

[105] "Intel Core I3-8109U Processor," https://ark.intel.com/content/www/us/en/ark/products/135936, 2018. 21, 77, 88

[106] "How Robots Change the World," https://resources.oxfordeconomics.com/how-robots-change-the-world, 2019. 1

[107] "GeForce GTX 1060," https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1060, 2021. 77

[108] "C++ Vector Class Library Version 2," https://www.agner.org/optimize/vcl_manual.pdf, 2022. 44, 61, 130

[109] M. Afrin, J. Jin, A. Rahman, A. Rahman, J. Wan, and E. Hossain, "Resource Allocation and Service Provisioning in Multi-Agent Cloud Robotics: A Comprehensive Survey," *IEEE Communications Surveys & Tutorials*, 2021. 110

[110] M. Anders, H. Kaul, S. Mathew, V. Suresh, S. Satpathy, A. Agarwal, S. Hsu, and R. Krishnamurthy, "2.9 TOPS/W Reconfigurable Dense/sparse Matrix-Multiply Accelerator with Unified INT8/INTI6/FP16 Datapath in 14nm Tri-Gate CMOS," in *IEEE Symposium on VLSI Circuits (VLSIC)*, 2018. 126, 133

[111] F. André, A.-M. Kermarrec, and N. Le Scouarnec, "Accelerated Nearest Neighbor Search with Quick Adc," in *International Conference on Multimedia Retrieval (ICMR)*, 2017. 122

[112] 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 on Computer Architecture (ISCA)*, 2022. 7, 68, 136

[113] M. Bakhshalipour and P. B. Gibbons, "Agents of Autonomy: A Systematic Study of Robotics on Modern Hardware," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2023. 7, 41, 135

[114] ——, "Agents of Autonomy: A Systematic Study of Robotics on Modern Hardware," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2024. 7, 41, 135

[115] ——, "Tartan: Microarchitecting a Robotic Processor," in *International Symposium on Computer Architecture (ISCA)*, 2024. 7, 108, 136

[116] M. Bakhshalipour, M. Likhachev, and P. B. Gibbons, "RTRBench: A Benchmark Suite for Real-Time Robotics," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022. 7, 21, 135

[117] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino Temporal Data Prefetcher," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2018. 83, 92

[118] M. Bakhshalipour, M. Qadri, D. Guri, S. B. Ehsani, M. Likhachev, and P. B. Gibbons, "Runahead A*: Speculative Parallelism for A* with Slow Expansions," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2023. 4, 7, 43, 59, 68, 136

[119] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2019. 83, 105, 130

[120] 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*, 2016. 4, 27, 43

[121] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv:1508.03619v4*, 2017. 132

[122] 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*, 2007. 4, 32, 43

[123] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively Parallelizing the RRT and the RRT*," in *International Conference on Intelligent Robots and Systems (IROS)*, 2011. 18, 53, 66

[124] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. Reddi, "MAVBench: Micro Aerial Vehicle Benchmarking," in *International Symposium on Microarchitecture (MICRO)*, 2018. 3, 11, 41, 42, 98

[125] B. Boroujerdian, H. Genc, S. Krishnan, B. P. Duisterhof, B. Plancher, K. Mansoorshahi, M. Almeida, W. Cui, A. Faust, and V. J. Reddi, "The Role of Compute in Autonomous Micro Aerial Vehicles: Optimizing for Mission Time and Energy Efficiency," *ACM Transactions on Computer Systems*, 2022. 9, 11

[126] C. Bowen and R. Alterovitz, "Closed-Loop Global Motion Planning for Reactive Execution of Learned Tasks," in *International Conference on Intelligent Robots and Systems (IROS)*, 2014. 10

[127] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-First Heuristic Search for Multicore Machines," *Journal of Artificial Intelligence Research*, 2010. 86, 104

[128] 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)*, 2020. 33

[129] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. Montiel, and J. D. Tardós, "ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial, and Multimap SLAM," *IEEE Transactions on Robotics*, 2021. 4, 43

[130] R. Canal, C. Hernandez, R. Tornero, A. Cilardo, G. Massari, F. Reghenzani, W. Fornaciari, M. Zapater, D. Atienza, A. Oleksiak *et al.*, "Predictive Reliability and Fault Management in Exascale Systems: State of the Art and Perspectives," *ACM Computing Surveys (CSUR)*, 2020. 109

[131] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011. 42

[132] L. Cayton, "Accelerating Nearest Neighbor Search on Manycore Systems," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2012. 122

[133] A. K. Chaudhary, R. Kothari, M. Acharya, S. Dangi, N. Nair, R. Bailey, C. Kanan, G. Diaz, and J. B. Pelz, "RITnet: Real-Time Semantic Segmentation of The Eye for Gaze Tracking," in *International Conference on Computer Vision Workshop (ICCVW)*, 2019. 10

[134] Y.-H. Chen and Y.-Y. Liu, "Dual-Addressing Memory Architecture for Two-Dimensional Memory Access Patterns," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013. 64

[135] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-Conscious Structure Definition," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999. 64

[136] C. Chung and C.-H. Yang, "A Distributed Autonomous and Collaborative Multi-Robot System Featuring a Low-Power Robot SoC in 22nm CMOS for Integrated Battery-Powered Minibots," in *International Solid-State Circuits Conference (ISSCC)*, 2019. 1, 43

[137] ——, "A 1.5-μJ/Task Path-Planning Processor for 2-D/3-D Autonomous Navigation of Microrobots," *IEEE Journal of Solid-State Circuits (JSSC)*, 2020. 1, 43, 53

[138] J. M. Domingos, N. Neves, N. Roma, and P. Tomás, "Unlimited Vector Extension with Data Streaming Support," in *International Symposium on Computer Architecture (ISCA)*, 2021. 115

[139] S. Durvasula, R. Kiguru, S. Mathur, J. Xu, J. Lin, and N. Vijaykumar, "VoxelCache: Accelerating Online Mapping in Robotics and 3D Reconstruction Tasks," *arXiv preprint arXiv:2210.08729*, 2022. 2

[140] P. H. E. Becker, J.-M. Arnau, and A. González, "KD Bonsai: ISA-Extensions to Compress KD Trees for Autonomous Driving Tasks," in *International Symposium on Computer Architecture (ISCA)*, 2023. 122, 123

[141] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2018. 125, 126

[142] C. Ericson, *Real-Time Collision Detection*, 2004. 53, 69

[143] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *International Symposium on Microarchitecture (MICRO)*, 2012. 27, 72, 81, 116, 129

[144] F. Farshidian, E. Jelavic, A. Satapathy, M. Giftthaler, and J. Buchli, "Real-Time Motion Planning of Legged Robots: A Model Predictive Control Approach," in *IEEE-RAS International Conference on Humanoid Robotics (Humanoids)*, 2017. 4, 34, 43, 57, 111

[145] Y. Feng, N. Goulding-Hotta, A. Khan, H. Reyserhove, and Y. Zhu, "Real-Time Gaze Tracking with Event-Driven Eye Segmentation," in *IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, 2022. 10

[146] 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)*, 2020. 2

[147] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012. 63

[148] D. Ferguson, N. Kalra, and A. Stentz, "Replanning with RRTs," in *International Conference on Robotics and Automation (ICRA)*, 2006. 4, 31, 43

[149] R. Firoozi, J. Tucker, S. Tian, A. Majumdar, J. Sun, W. Liu, Y. Zhu, S. Song, A. Kapoor, K. Hausman *et al.*, "Foundation Models in Robotics: Applications, Challenges, and the Future," *arXiv preprint arXiv:2312.07843*, 2023. 14

[150] W. Fornaciari, G. Agosta, D. Atienza, C. Brandolese, L. Cammoun, L. Cremona, A. Cilardo, A. Farres, J. Flich, C. Hernandez *et al.*, "Reliable Power and Time-Constraints-Aware Predictive Management of Heterogeneous Exascale Systems," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2018. 109

[151] C. S. Gadde, M. S. Gadde, N. Mohanty, and S. Sundaram, "Fast Obstacle Avoidance Motion in Small Quadcopter Operation in a Cluttered Environment," in *International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2021. 4, 43, 59, 111

[152] 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)*, 2014. 4, 30, 43

[153] R. Geraerts and M. H. Overmars, "Creating High-Quality Paths for Motion Planning," *The International Journal of Robotics Research*, 2007. 4, 43

[154] R. Ghzouli, S. Dragule, T. Berger, E. B. Johnsen, and A. Wasowski, "Behavior Trees and State Machines in Robotics Applications," *arXiv preprint arXiv:2208.04211*, 2022. 4, 43, 55, 111

[155] 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)*, 2019. 109

[156] N. Gupta and D. S. Nau, "On the Complexity of Blocks-World Planning," *Artificial Intelligence*, 1992. 32

[157] 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)*, 2014. 24, 128

[158] 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*, 1968. 21, 25, 70, 116

[159] Y. Hu, Q. Xie, V. Jain, J. Francis, J. Patrikar, N. Keetha, S. Kim, Y. Xie, T. Zhang, Z. Zhao *et al.*, "Toward General-Purpose Robots Via Foundation Models: A Survey and Meta-Analysis," *arXiv preprint arXiv:2312.08782*, 2023. 14

[160] L. Huang, Y. Gong, Y. Sui, X. Zang, and B. Yuan, "MOPED: Efficient Motion Planning Engine with Flexible Dimension Support," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2024. 18, 30, 53

[161] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation Learning: A Survey of Learning Methods," *ACM Computing Surveys (CSUR)*, 2017. 33, 59

[162] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison, "RLBench: The Robot Learning Benchmark & Learning Environment," *IEEE Robotics and Automation Letters*, 2020. 3, 20, 21, 41, 42, 44, 46

[163] N. Jaquier, L. Rozo, S. Calinon, and M. Bürger, "Bayesian Optimization Meets Riemannian Manifolds in Robot Learning," in *Conference on Robot Learning*, 2020. 4, 36, 43

[164] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A Scalable Architecture for Ordered Parallelism," in *International Symposium on Microarchitecture (MICRO)*, 2015. 18, 26

[165] M. Kar, A. Agarwal, S. Hsu, D. Moloney, G. Chen, R. Kumar, H. Sumbul, P. Knag, M. Anders, H. Kaul, J. Byrne, L. Sarti, R. Krishnamurthy, and V. De, "A Ray-Casting Accelerator in 10nm CMOS for Efficient 3D Scene Reconstruction in Edge Robotics and Augmented Reality Applications," in *IEEE Symposium on VLSI Circuits (VLSIC)*, 2020. 1, 127

[166] 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*, 1996. 28, 80

[167] 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)*, 2013. 24, 54

[168] G. Keramidas, M. Mavropoulos, A. Karvouniari, and D. Nikolos, "Instruction Based Management of Faulty Data Caches." 64

[169] A. Kerr, G. Diamos, and S. Yalamanchili, "A Characterization and Analysis of GPGPU Kernels," Georgia Institute of Technology, Tech. Rep., 2009. 53

[170] F. Kherif and A. Latypova, "Principal Component Analysis," in *Machine Learning*, 2020. 128

[171] Y. Kim, D. Shin, J. Lee, Y. Lee, and H.-J. Yoo, "A 0.55V 1.1mW Artificial-Intelligence Processor with PVT Compensation for Micro Robots," in *International Solid-State Circuits Conference (ISSCC)*, 2016. 1, 43

[172] A. Kishimoto, A. Fukunaga, and A. Botea, "Scalable, Parallel Best-First Search for Optimal Sequential Planning," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2009. 86, 94, 104

[173] S. Koppula, L. Orosa, A. G. Yağlıkçı, R. Azizi, T. Shahroodi, K. Kanellopoulos, and O. Mutlu, "EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM," in *International Symposium on Microarchitecture (MICRO)*, 2019. 115

[174] R. E. Korf, "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, 1985. 4, 43

[175] L. Koutras and Z. Doulgeri, "Dynamic Movement Primitives for Moving Goals with Temporal Scaling Adaptation," in *International Conference on Robotics and Automation (ICRA)*, 2020. 4, 33, 43, 59, 111

[176] H. Koyuncu, "Loss Function Selection in NN Based Classifiers: Try-Outs with a Novel Method," in *International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, 2020. 128

[177] S. Krishnan, Z. Wan, K. Bhardwaj, N. Jadhav, A. Faust, and V. J. Reddi, "Roofline Model for UAVs: A Bottleneck Analysis Tool for Onboard Compute Characterization of Autonomous Unmanned Aerial Vehicles," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022. 11

[178] Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J. P. How, "Real-Time Motion Planning with Applications to Autonomous Urban Driving," *IEEE Transactions on control systems technology*, 2009. 81

[179] S. M. LaValle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," 1998. 21, 29, 43, 53, 80, 111, 112

[180] U. Lee, J. Jung, S. Shin, Y. Jeong, K. Park, D. H. Shim, and I.-s. Kweon, "EureCar Turbo: A Self-Driving Car That Can Handle Adverse Weather Conditions," in *International Conference on Intelligent Robots and Systems (IROS)*, 2016. 10

[181] J. J. Leonard, D. A. Mindell, and E. L. Stayton, "Autonomous Vehicles, Mobility, and Employment Policy: The Roads Ahead," *Massachusetts Institute of Technology, Cambridge, MA, Rep. RB02-2020*, 2020. 1

[182] 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)*, 2018. 18, 28, 30, 53, 66, 67, 104

[183] M. Likhachev, D. I. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A*: An Anytime, Replanning Algorithm." in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2005. 116, 118

[184] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The Architectural Implications of Autonomous Driving: Constraints and Acceleration," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018. 3, 41, 42

[185] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *European Conference on Computer Vision (ECCV)*, 2014. 51

[186] W. Liu, B. Yu, Y. Gan, Q. Liu, J. Tang, S. Liu, and Y. Zhu, "Archytas: A Framework for Synthesizing and Dynamically Optimizing Accelerators for Robotic Localization," in *International Symposium on Microarchitecture (MICRO)*, 2021. 2

[187] 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)*, 2020. 27, 64, 107

[188] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-Out Processors," in *International Symposium on Computer Architecture (ISCA)*, 2012. 78, 137

[189] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration," in *International Symposium on Computer Architecture (ISCA)*, 2016. 27, 117, 118

[190] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra, "Benchmarking Reinforcement Learning Algorithms on Real-World Robots," in *Conference on Robot Learning*, 2018. 46

[191] V. Mayoral-Vilches, J. Jabbour, Y.-S. Hsiao, Z. Wan, A. Martínez-Fariña, M. Crespo-Álvarez, M. Stewart, J. M. Reina-Muñoz, P. Nagras, G. Vikhe, M. Bakhshalipour, M. Pinzger, S. Rass, S. Panigrahi, G. Corradi, N. Roy, P. B. Gibbons, S. M. Neuman, B. Plancher, and V. J. Reddi, "RobotPerf: An Open-Source, Vendor-Agnostic, Benchmarking Suite for Evaluating Robotics Computing System Performance," in *International Conference on Robotics and Automation (ICRA)*, 2024. 11, 109

[192] X. Meng, H. Wang, and B. Liu, "A Robust Vehicle Localization Approach Based on GNSS/IMU/DMI/LIDAR Sensor Fusion for Autonomous Vehicles," *Sensors*, 2017. 59

[193] A. K. Menon, A. S. Rawat, S. J. Reddi, and S. Kumar, "Can Gradient Clipping Mitigate Label Noise?" in *International Conference on Learning Representations (ICLR)*, 2019. 119

[194] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu *et al.*, "Learning to Navigate in Complex Environments," *arXiv preprint arXiv:1611.03673*, 2016. 4, 43

[195] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping That Provably Converges," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003. 4, 43

[196] S. Moore, "3 New Chips to Help Robots Find Their Way Around," *IEEE Spectrum*, 2019. 66, 110

[197] 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)*, 2018. 107, 123, 126

[198] S. P. Muralidhara, M. Kandemir, and P. Raghavan, "Intra-Application Cache Partitioning," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010. 125

[199] M. P. Muresan, I. Giosan, and S. Nedevschi, "Stabilization and Validation of 3D Object Position Using Multimodal Sensor Fusion and Semantic Segmentation," *Sensors*, 2020. 56, 111

[200] 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)*, 2016. 2, 9, 10, 28, 53, 67, 104

[201] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A Modern Primer on Processing in Memory," *arXiv preprint arXiv:2012.03112*, 2020. 24

[202] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector Runahead," in *International Symposium on Computer Architecture (ISCA)*, 2021. 115

[203] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled Vector Runahead," in *International Symposium on Microarchitecture (MICRO)*, 2023. 115

[204] C. G. Nevill-Manning and I. H. Witten, "Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm," *Journal of Artificial Intelligence Research*, 1997. 92

[205] 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*, 2011. 24

[206] X. Ni, L. Fang, and H. Huttunen, "Adaptive L2 Regularization in Person Re-Identification," in *International Conference on Pattern Recognition (ICPR)*, 2021. 119

[207] 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 on Computer Architecture (ISCA)*, 2021. 24

[208] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-Dataflow Acceleration," in *International Symposium on Computer Architecture (ISCA)*, 2017. 34

[209] H. Ohta, N. Akai, E. Takeuchi, S. Kato, and M. Edahiro, "Pure Pursuit Revisited: Field Testing of Autonomous Vehicles in Urban Areas," in *International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2016. 4, 43, 51, 111

[210] O. A. Osman, M. Hajij, P. R. Bakhit, and S. Ishak, "Prediction of Near-Crashes from Observed Vehicle Kinematics Using Machine Learning," *Transportation Research Record*, 2019. 120

[211] A. Pajuelo, A. González, and M. Valero, "Speculative Dynamic Vectorization," *ACM SIGARCH Computer Architecture News*, 2002. 115

[212] 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*, 2020. 4, 35, 43

[213] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, "The Volumepro Real-Time Ray-Casting System," in *Proceedings of the Annual Conference on computer Graphics and Interactive Techniques*, 1999. 127

[214] M. Phillips, M. Likhachev, and S. Koenig, "PA*SE: Parallel A* for Slow Expansions," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2014. 18, 67, 70, 86, 87, 94, 105

[215] G. Pinto, F. Castor, and Y. D. Liu, "Understanding Energy Behaviors of Thread Management Constructs," in *International Conference on Object Oriented Programming Systems Languages & Applications*, 2014. 112

[216] I. Pohl, "Heuristic Search Viewed As Path Finding in a Graph," *Artificial Intelligence*, 1970. 4, 10, 27, 43, 57, 84, 100, 111, 113, 116

[217] N. A. Radford, P. Strawser, K. Hambuchen, J. S. Mehling, W. K. Verdeyen, A. S. Donnan, J. Holley, J. Sanchez, V. Nguyen, L. Bridgwater *et al.*, "Valkyrie: Nasa's First Bipedal Humanoid Robot," *Journal of Field Robotics*, 2015. 109

[218] M. Roberts, J. Ramapuram, A. Ranjan, A. Kumar, M. A. Bautista, N. Paczan, R. Webb, and J. M. Susskind, "Hypersim: A Photorealistic Synthetic Dataset for Holistic Indoor Scene Understanding," in *International Conference on Computer Vision (ICCV)*, 2021. 47, 54, 55

[219] 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)*, 2013. 34

[220] 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 on Computer Architecture (ISCA)*, 2018. 2, 9, 34, 48, 58

[221] 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. 20, 21, 38, 44

[222] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-Based Approximation for Data Parallel Applications," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014. 118

[223] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," in *International Symposium on Computer Architecture (ISCA)*, 2011. 125, 126

[224] ——, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *International Symposium on Computer Architecture (ISCA)*, 2013. 21, 37, 44, 77, 111

[225] C. Schulz and A. Zell, "Real-Time Graph-Based SLAM with Occupancy Normal Distributions Transforms," in *International Conference on Robotics and Automation (ICRA)*, 2020. 4, 43

[226] D. Shah, N. Yang, and T. M. Aamodt, "Energy-Efficient Realtime Motion Planning," in *International Symposium on Computer Architecture (ISCA)*, 2023. 18, 30, 48, 53

[227] 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)*, 2018. 35

[228] S. Shen, Y. Cai, J. Qiu, and G. Li, "Dynamic Dense RGB-D SLAM Using Learning-Based Visual Odometry," *arXiv preprint arXiv:2205.05916*, 2022. 111, 113

[229] 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)*, 2015. 26, 67, 83

[230] Y. Shi, W. Zhang, Z. Yao, M. Li, Z. Liang, Z. Cao, H. Zhang, and Q. Huang, "Design of a Hybrid Indoor Location System Based on Multi-Sensor Fusion for Robot Navigation," *Sensors*, 2018. 120

[231] E. Shipovalov and V. Pryanichnikov, "Scalable State Space Search on the GPU with Multi-Level Parallelism," in *International Symposium on Parallel and Distributed Computing (ISPDC)*, 2020. 70

[232] B. Shoshany, "A C++17 Thread Pool for High-Performance Scientific Computing," *arXiv e-prints*, 2021. 93

[233] J. Siderska, "Robotic Process Automation—a Driver of Digital Transformation?" *Engineering Management in Production and Services*, 2020. 110

[234] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, "Manipulation Planning with Probabilistic Roadmaps," *The International Journal of Robotics Research*, 2004. 4, 43

[235] J. Slaney and S. Thiébaux, "Blocks World Revisited," *Artificial Intelligence*, 2001. 32

[236] T. Song, W. Rim, J. Jung, G. Yang, J. Park, S. Park, Y. Kim, K.-H. Baek, S. Baek, S.-K. Oh *et al.*, "A 14 nm FinFET 128 Mb SRAM With V_{MIN} Enhancement Techniques for Low-Power Applications," *IEEE Journal of Solid-State Circuits*, 2014. 126, 133

[237] N. R. Sturtevant, "Benchmarks for Grid-Based Pathfinding," *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 2012. 25, 78, 93, 98

[238] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P. Topham, and B. Franke, "Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMPs," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2012. 126

[239] S. Thrun, "Probabilistic Robotics," *Communications of the ACM*, 2002. 4, 43

[240] 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*, 2020. 4, 23, 43, 50, 111

[241] R. Vang-Mata, *Multilayer Perceptrons: Theory and Applications*, 2020. 118

[242] K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell, "Molecular Caches: A Caching Structure for Dynamic Creation of Application-Specific Heterogeneous Cache Regions," in *International Symposium on Microarchitecture (MICRO)*, 2006. 126

[243] D. Vrontis, M. Christofi, V. Pereira, S. Tarba, A. Makrides, and E. Trichina, "Artificial Intelligence, Robotics, Advanced Technologies and Human Resource Management: A Systematic Review," *The international journal of human resource management*, 2022. 14

[244] Z. Wan, B. Yu, T. Y. Li, J. Tang, Y. Zhu, Y. Wang, A. Raychowdhury, and S. Liu, "A Survey of FPGA-Based Robotic Computing," *arXiv preprint arXiv:2009.06034*, 2020. 109

[245] Y. Wang, L. Zhang, and G. Chen, "Optimal Sensor Placement for Obstacle Detection of Manipulator Based on Relative Entropy," in *IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 2019. 111, 112

[246] A. Weinstock and R. Holladay, "Parallel A* Graph Search," Project Report, School of Computer Science, Carnegie Mellon University, available in https://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf, 2017. 96

[247] J. T. Wen and S. H. Murphy, "PID Control for Robot Manipulators," 1990. 4, 43, 53, 111

[248] T. F. Wenisch, "Temporal Memory Streaming," Ph.D. dissertation, Carnegie Mellon University, 2007. 92

[249] T. Whelan, S. Leutenegger, R. Salas-Moreno, B. Glocker, and A. Davison, "ElasticFusion: Dense SLAM without a Pose Graph," 2015. 4, 43

[250] V. A. Ying, M. C. Jeffrey, and D. Sanchez, "T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware," in *International Symposium on Computer Architecture (ISCA)*, 2020. 105

[251] Z. Ying, S. Bhuyan, Y. Kang, Y. Zhang, M. T. Kandemir, and C. R. Das, "EdgePC: Efficient Deep Learning Analytics for Point Clouds on Edge Devices," in *International Symposium on Computer Architecture (ISCA)*, 2023. 122

[252] A. Younis, L. Shixin, S. Jn, and Z. Hai, "Real-Time Object Detection Using Pre-Trained Deep Learning Models MobileNet-SSD," in *International Conference on Computing and Data Engineering (ICCDE)*, 2020. 4, 43, 51, 111, 112, 120

[253] 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)*, 2020. 3, 9, 15, 41, 42, 46, 48, 54, 66, 117

[254] 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*, 2019. 4, 22, 43, 48, 111, 112

[255] Y. Zhou, "Open Source Protein REdesign for You on a GPU," https://github.com/zhou13/gOSPREY, 2015. 94

[256] Y. Zhou and J. Zeng, "Massively Parallel A* Search on a GPU," in *AAAI Conference on Artificial Intelligence*, 2015. 4, 18, 43, 59, 94, 116

[257] Y. Zhu, "RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2022. 120