RISC-V Architecture for Motion Planning Algorithms in Autonomous Drone Applications

A senior design project submitted in partial fulfillment of the requirements for the degree of Bachelor of Science at Harvard University

Anthony J.W. Kenny

S.B. Candidate in Electrical Engineering

Faculty Advisor: Vijay Janapa Reddi

Harvard University School of Engineering and Applied Sciences

Cambridge, MA

March 1st, 2020

Version: 4.1

Abstract

This thesis demonstrates the design of RISC-V computer architecture that supports faster execution of motion planning algorithms for drone applications. First, it shows the analysis of computational performance of Rapidly-exploring Random Tree (RRT), a sampling-based motion planning algorithm commonly used in autonomous drones. Having identified collision detection as the biggest area of opportunity for improved performance, it describes the process of designing specialized hardware, taking advantage of parallelization, that quickly detects collisions. Finally, it presents how this specialized functional unit can be implemented in a processor, and the RISC-V Instruction Set Architecture (ISA) extended to invoke execution to massively reduce the execution time of collision detection.

Contents

List of Acronyms

API Application Programming Interface

FPGA Field Programmable Gate Array

CPU Central Processing Unit

GPU Graphics Processing Unit

GUI Graphical User Interface

ISA Instruction Set Architecture

RRT Rapidly-exploring Random Tree

RTOS Real-Time Operating Systems

2D 2-Dimensional

3D 3-Dimensional

Use glossary package

Use better acronym package that includes plurals

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Summary

1.1.1 Background

TODO: Summarize the following points:
1) Need for faster execution of motion planning in drones
2) Strategy of specialized hardware
3) RISC-V ISA and its potentials including extendibility

Robotics

For well over 2000 years, the concept of robotics, albeit not always with such a term, has fascinated humans. As early as the first century A.D., the Greek mathematician and engineer, Heron of Alexandria, described more than 100 different machines and automata in *Pneumatica* and *Automata*[1]. In 1898, Nikola Tesla demonstrated the first radio-controlled vessel. Since then, the world has seen widespread application of robotics in manufacturing, mining, transport, exploration, and weaponry. For the last few decades, robots have operated in controlled, largely unchanging environments (e.g. an assembly line) where their environment and movements are largely known *a priori*.
However, in recent years a new generation of autonomous robots has been developed for a wide range of real-world, complex applications. The increasing trend the use of autonomous robots is shown in Figure 1.1. These new robots, unlike those traditional ones described above, are required to adapt to the changing environment in which they operate. As such, they must perform motion planning in real time.

Some sort of line/bar graph showing the increasing use of Autonomous robots over time. Need to find

Figure 1.1: The use of Autonomous Robots over time

Motion Planning

TODO: More of an introduction to motion planning.

Motion Planning refers to the problem of determining how a robot moves through a space to acheive a goal. Chapter 2 provides a detailed explanation of motion planning and of RRT, a commonly used motion planning algorithm.

On the algorithmic level, motion planning has been extensively studied and many solutions exist. However, current algorithms running on regular Central Processing Unit (CPU)s are too slow to execute in real time for robots operating in complex environments. Simply solving this problem with more raw computing power, using energy hungry Graphics Processing Unit (GPU)s may have merit in tethered robots. On the other hand, untethered applications, such as autonomous drones, where limiting power consumption is a primary concern, this strategy is infeasible.

Hardware Acceleration

Specialized hardware designed to perform specific functions can yield significantly higher performance than software running on general purpose processors, and lower power consumption than GPUs.

More detail here. Reference prior work

RISC-V

TODO: Introduction to RISC-V and its merits in this problem

1.1.2 Problem Definition

Problem Statement

Revise problem statement

Current processors cannot compute motion planning algorithms quickly enough for robots to operate in high complexity environments. Autonomous drones are a specific case of robots requiring real-time motion planning in complex environments. The state-of-the-art strategy of using a Graphics Processing Unit (GPU) to accelerate the execution of these algorithms requires too much power to be cost-effective or feasible for drones to sustain flight for useful periods of time.

End User

The end user of this project is a developer of autonomous drones. Such developers have a need for computing hardware that executes motion planning algorithms faster and more power efficiently than existing methods. This thesis will provide a processor design that is synthesizable on an Field Programmable Gate Array (FPGA), giving developers a processer for which a Real-Time Operating Systems (RTOS), or bare metal code, can be written. Additionally, these developers have a requirement that using a new processor for a drone will not require a massive investment in re-development. As such, this thesis will provide the toolchain necessary to compile C code into executable instructions on the new processor.

TODO: Revise End User

1.2 Prior Work

TODO: Summary of prior work

1.3 Project Overview

1.3.1 Proposed Solution

This thesis proposes aims to provide drone developers

Proposed Solution

1.3.2 Project Specifications

Project Specifications

1.3.3 Project Structure

Project Structure/Timeline

Chapter 2

Motion Planning in Software

2.1 Background

Motion Planning Algorithms refer to the set of algorithms that find possible sequences of valid configurations for a robot in a space.

TODO: Background on Motion Planning Algorithms.

2.2 Rapidly-Exploring Random Tree

RRT is an algorithm designed to efficiently search, and thus plan a path through, a high-complexity environment by randomly sampling points and building a tree. The algorithm randomly samples points, draws an edge from the nearest currently existing node in the tree, to grow the tree in the space. It is inherently biased to grow towards large unsearched areas of the problem. RRT was developed by S. LaVelle[2] and J. Kuffner[3]. It is used in autonomous robotic motion planning problems such as autonomous drones, the focus of this thesis.

2.2.1 Algorithm

Building the Tree

Put simply, RRT builds a tree (referred to as a graph) of possible configurations, connected by edges, for a robot of some physical description. It does so by randomly sampling the configuration space and adding configurations to the graph. From this graph, a path from the initial configuration to some goal configuration can be found, given a high enough number of iterations. As such, RRT can be considered probabalistically complete. The pseudo-code for RRT can be seen in Algorithm 2.1

Algorithm 2.1: Rapidly-Exploring Random Tree in Free Configuration Space

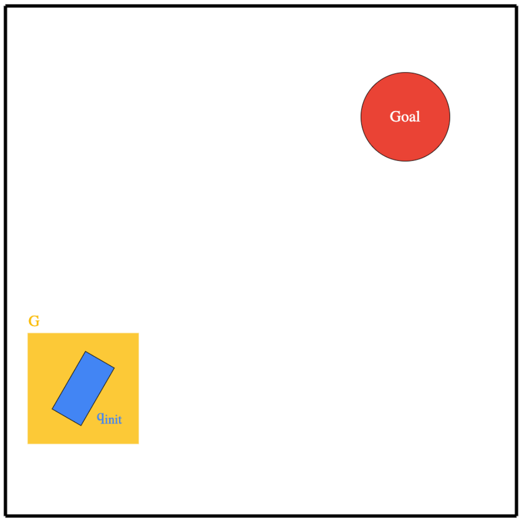
Inputs: Initial configuration q_{init},
Number of nodes in graph K,
Incremental Distance Δq

Output: RRT Graph G with K configurations $[q]$ & edges $[e]$

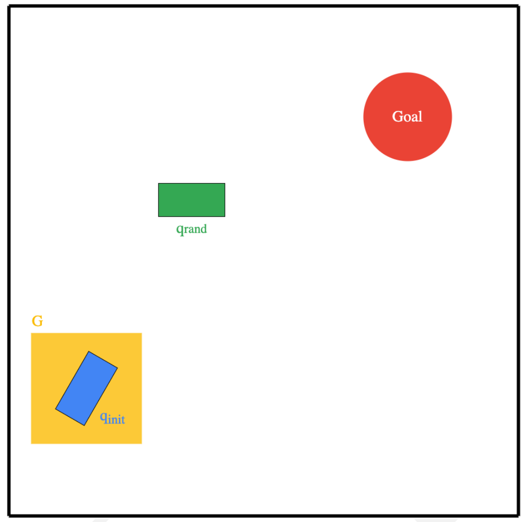
G.init();
for $k = 1$ to K **do**
  $q_{rand} \leftarrow$ randomConfiguration();
  $q_{near} \leftarrow$ nearestVertex(q_{rand}, G);
  $q_{new} \leftarrow$ newVertex(q_{near}, q_{rand}, Δq);
  G.addVertex(q_{new});
  G.addEdge(q_{near}, q_{new});
end

Algorithm 2.1 can be visually represented in Figure 2.1. Consider a 2-Dimensional (2D) robot operating in a 2D workspace. A Graph G is initialized containing an initial configuration, q_{init}, with constraints on the number of nodes that the graph can hold, K, and the maximum distance between two nodes, Δq. This is shown in Sub-figure 2.1a. A random configuration for the robot, q_{rand} is generated (2.1b). The nearest existing configuration in G, q_{near}, is found. (In the first iteration, $q_{near} = q_{init}$, shown in Sub-figure

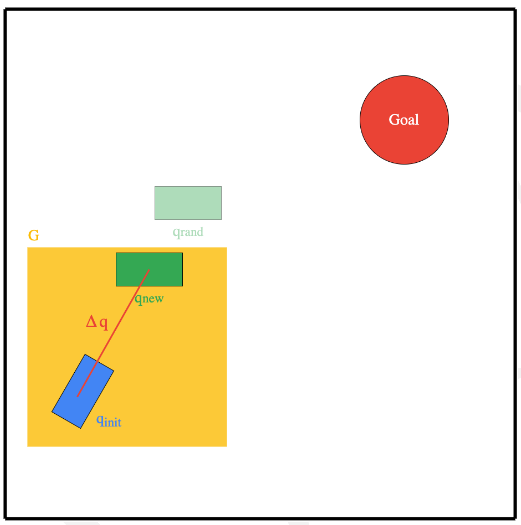
2.1c). The distance between q_{near} and q_{rand} is calculated. If this distance is less than Δq, $q_{new} = q_{rand}$. If not, q_{new} is selected, typically by moving by Δq from q_{near} towards q_{rand} (2.1c). q_{new} is then added to G. This is repeated for K configurations.



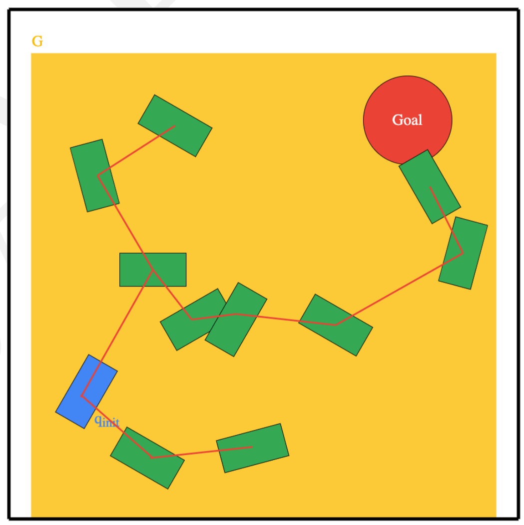
(a) Graph G contains only q_{init}



(b) The first random configuration, q_{rand}, is generated



(c) In first iteration, $q_{near} = q_{init}$. Distance between q_{init} and q_{rand} is greater than Δq, so q_{new} is generated and added to G



(d) This is repeated K times. For G, $K = 10$, and the red line represents the edges between configurations

Figure 2.1: Step by step demonstration of RRT Algorithm for 2D robot in 2D space

Collision Detection

Algorithm 2.1 shows how RRT builds a graph of possible configurations connected by edges in a free configuration space. However, in real-world applications, a robot's configuration space often contains obstacles. As such, collision detection must be included in the algorithm. The two types of collisions the algorithm must check for are *configuration collisions* (those where the robot would collide with an obstacle in a given configuration) and *edge collisions* (where the robot would collide when moving between two collision free configurations).

The RRT with configuration and edge collision detection can be seen in Algorithm 2.2. The method of implementing RRT with collision detection to model a drone in 3D space is detailed in Section 2.2.2.

Algorithm 2.2: Rapidly-Exploring Random Tree with Collision Detection

---

Inputs: Initial configuration q_{init},
Number of nodes in graph K,
Incremental Distance Δq,
Space S containing obstacles
Output: RRT Graph G with K configurations $[q]$ & edges $[e]$

G.init();
for $k = 1$ to K **do**
  while !pointCollision(q_{new}) **do**
    $q_{rand} \leftarrow$ randomConfiguration();
    $q_{near} \leftarrow$ nearestVertex(q_{rand}, G);
    $q_{new} \leftarrow$ newVertex(q_{near}, q_{rand}, Δq);
  end
  $e_{new} \leftarrow$ newEdge(q_{near}, q_{new})
  if !edgeCollision(e_{new}) **then**
    G.addVertex(q_{new});
    G.addEdge(q_{near}, q_{new});
  else
    $k = k - 1$;
  end
end

2.2.2 Implementation

With RRT selected as the benchmark algorithm against which to test specialised hardware, this project required an implementation of the algorithm that satisfied the following criteria.

Better RRT Implementation introductory sentence

Requirement	Description and Justification
C/C++ Implementation	As outlined in Section 1.3.3, the critical step in determining the design of specialized hardware to accelerate RRT is CPU performance analysis of the algorithm to determine computational hot-spots. Implementations in C allow for the use of certain CPU profiling tools, described in Section 2.2.3, unlike higher-level languages such as Python.
Models Drone as Point	In reality, implementing RRT for a drone would model the robot as a 3-Dimensional (3D) object defined by coordinates $\{x, y, z\}$ and Euler angles $\{\alpha, \beta, \gamma\}$. However, for simplicity's sake, modelling the drone as a point defined by coordinates $\{x, y, z\}$ will suffice. Time permitting, this could be revisited. Change this based on whether time does permit
Mirrors Algorithm	In order for the results of CPU performance analysis to be easy to understand, software implementation of RRT should call functions that mirror the functions described in Algorithms 2.1 and 2.2.

Table 2.1: Technical Specifications for RRT Implementation

Improve this table

The original intention was to find an existing implementation of RRT that could fulfill these requirements. Most open source implementations found online were in Python, and all those implemented in C were unsuitable[4][5][6][7], as they had extraneous GUIs, reliance on external Application Programming Interface (API)s, and other features that would distort analysis of algorithmic hot-spots.

As a result, it was necessary to build a C implementation of RRT from the ground up that satisfied the requirements in Table 2.1. It can be found in this project's GitHub repository. It follows Algorithm 2.1 closely. For monitoring correctness, I build in an optional GUI that shows the tree, starting node, and obstacles.

Implementation in 2D

The first step was to implement RRT with a 2-Dimensional workspace.

More detail

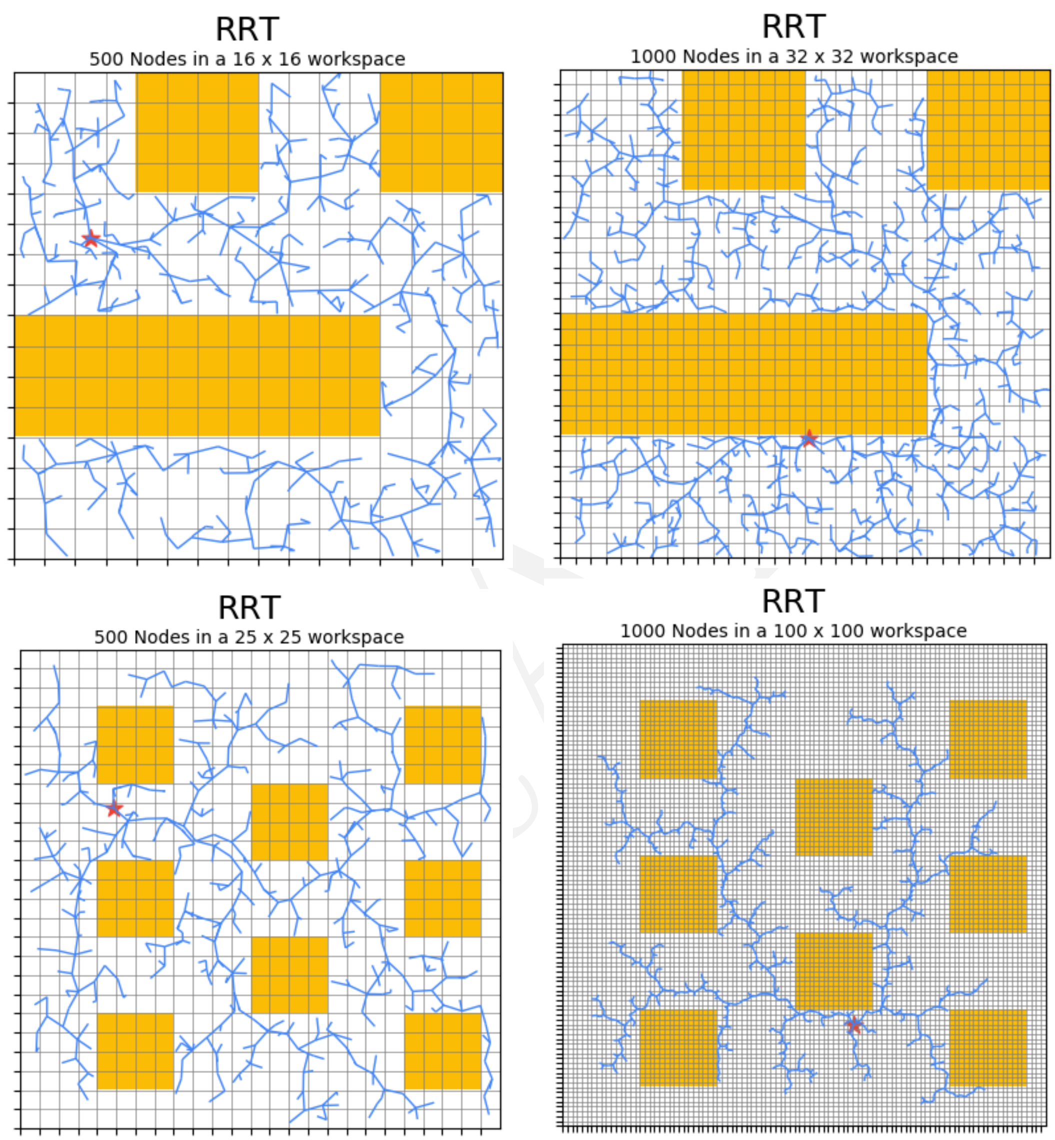


Figure 2.2: 2D RRT Implementation shown by GUI

Implementation in 3D

Describe implementation in 3D

Figure 2.3: 3D RRT Implementation shown by GUI

2.2.3 Performance Analysis

To restate, the aim of this thesis is to design a computer processor with reduced execution time of motion planning algorithms, such as RRT. As such, it is important to understand the elements of the algorithm that have the highest percentage of CPU execution time. To determine this, it was necessary to implement my own, naive but typical, RRT in C. This program could then be compiled and analysed using a software performance profiling tool. With this, I could design experiments to determine the critical RRT functions (those occupying a majority of CPU time) and see how this varies given different parameters.

Introduction to purpose of analysis and methods of doing so. Something better than the above

VTune Amplifier Analysis

VTune Amplifier performance profiler is an application for software performance analysis. It provides functionality to examine hot-spots for CPU execution time through a top down analysis, shown below in Figure 2.4. As can be seen from the figure, the top down analysis tool shows the percentage of CPU time taken up by each function. I used this tool to profile the algorithm's performance as I changed certain parameters.

Rewrite the above

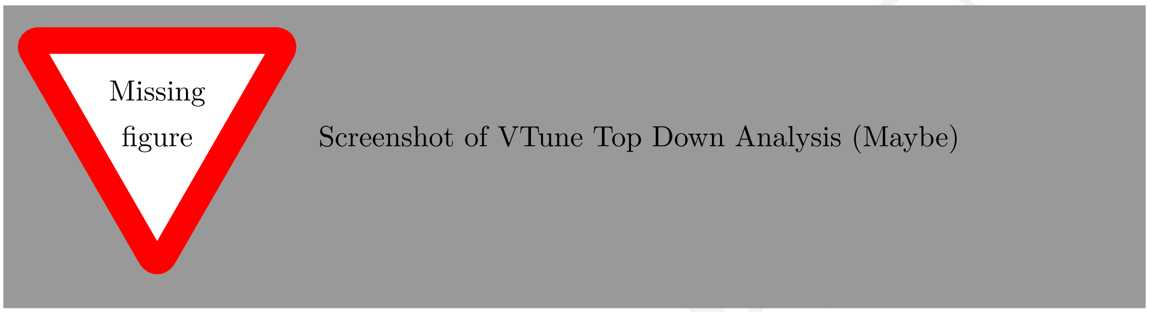


Figure 2.4: VTune Amplifier TopDown Analysis Example

Internal Timing Analysis

Comparison

2.3 Improving Performance

Chapter 3

Motion Planning in Hardware

3.1 Background to Hardware Optimization

3.2 HoneyBee

3.2.1 Design

3.2.2 Build

3.2.3 Measurement and Analysis

3.2.4 Iterations

Dimensions	Mac OS	Ubuntu	1	2	3	4
4x4x4	2	2	21.6	1.5	0.44	0.47
8x8x8	23	19	151	5.53	2.2	1.79
16x16x16	166	180	1133	41.37	13.08	12.11
32x32x32	1317	1424	8783	328	103	104

Table 3.1: Simulated performance of HB-A in microseconds

Chapter 4

RISC-V Processor

4.1 RISC-V ISA

4.2 Extending RISC-V

4.3 PhilosophyV

Chapter 5

Conclusion

5.1 Discussion of Results

Bibliography

[1] H. (Alexandrinus), *De gli automati, overo machine se moventi, Volume 2.*

[2] S. M. LaValle, “Rapidly-Exploring Random Trees: A New Tool for Path Planning,” *In*, vol. 129, pp. 98–11, 1998.

[3] S. M. LaValle and J. J. Kuffner, “Randomized kinodynamic planning,” *International Journal of Robotics Research*, vol. 20, pp. 378–400, may 2001.

[4] RoboJackets, “RRT,” 2019.
`https://github.com/RoboJackets/rrt`.

[5] M. Planning, “rrt-algorithms,” 2019.
`https://github.com/motion-planning/rrt-algorithms`.

[6] Sourishg, “rrt-simulator,” 2017.
`https://github.com/sourishg/rrt-simulator`.

[7] Vss2sn, “Path Planning,” 2019.
`https://github.com/vss2sn/path{_}planning`.

Appendices

Appendix A

Project Repository

TODO: Provide information about this project's repository

Todo list