if goingToCrash()
dont();

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

Preface **i**
Abstract i
Table of Contents ii
List of Acronyms iv
List of Algorithms v
List of Figures vi
List of Tables vii

1 Introduction **1**
1.1 Problem Summary 1
1.1.1 Background 1
1.1.2 Problem Definition 4
1.2 Prior Work 4
1.3 Project Overview 4
1.3.1 Proposed Solution 4
1.3.2 Project Specifications 4
1.3.3 Project Structure 5

2 Motion Planning in Software **6**
2.1 Background 6
2.2 Rapidly-Exploring Random Tree 7
2.2.1 Algorithm 7
2.2.2 Implementation 10
2.3 Performance Analysis 13
2.3.1 Methodology 13
2.3.2 Results 14

3 Motion Planning in Hardware **20**
3.1 Background to Hardware Optimization 20
3.2 HoneyBee 20
3.2.1 Design 20

3.2.2 Build 20
3.2.3 Measurement and Analysis 20
3.2.4 Iterations 20

4 RISC-V Processor **22**
4.1 RISC-V ISA 22
4.2 Extending RISC-V 22
4.3 PhilosophyV 22

5 Conclusion **23**
5.1 Discussion of Results 23

Bibliography **24**

Appendices **25**

A Project Repository **26**

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
UAV	Unmanned Aerial Vehicle
2D	2-Dimensional
3D	3-Dimensional

Use glossary package

Use better acronym package that includes plurals

List of Algorithms

2.1 Rapidly-Exploring Random Tree in Free Configuration Space 7
2.2 Rapidly-Exploring Random Tree with Collision Detection 9

List of Figures

1.1 The use of Autonomous Robots over time 2
1.2 Simple Visualization of Computer Implementation Hierarchy 3

2.1 Step by step demonstration of RRT Algorithm for 2D robot in 2D space . . 8
2.2 2D RRT Implementation shown by Graphical User Interface (GUI) 11
2.3 3D RRT Implementation shown by GUI 12
2.4 VTune Amplifier TopDown Analysis Example 13
2.5 RRT Functions as a % of Total CPU Exectution Time 15
2.5 RRT Functions as a % of Total CPU Exectution Time (cont.) 16
2.6 RRT Functions Exectution Time, with Bucket Optimization 18

List of Tables

2.1 Technical Specifications for RRT Implementation 10
2.2 Comparison of Timing Methods 14

3.1 Simulated performance of HB-A in microseconds 21

Chapter 1

Introduction

1.1 Problem Summary

1.1.1 Background

The Unmanned Aerial Vehicle (UAV) has been utilised in military applications extensively throughout the late 20th and early 21st century. However, over the last decade, their use in non-military uses, such as commerical, scientific, agricultural, and recreational, such that the number of civilian drones vastly outnumber military UAVs.Particularly in the commercial sector, such rapid growth in the number and range of applications means that autonomy is key for the profitable adoption of UAVs. Such autonomy relies on efficient computation of motion planning algorithms. However, the implementation of these algorithms can be quite computationally expensive, and thus slow and/or detrimentally power consuming. As such, this thesis aims to design specialized hardware to more efficiently compute motion plans for autonomous drones.

cite

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

Missing figure

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

above, are required to adapt to the changing environment in which they operate. As such, they must perform motion planning in real 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

Hardware acceleration refers to the strategy of using computer hardware specifically designed to execute a function more efficiently than can be achieved by software running on a general purpose CPU. 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.

To take a step back, consider a typical computer implementation hierarchy, demonstrated in Figure 1.2. **User level applications**, such as Google Chrome, Microsoft Word, and Apple's iTunes, sit at the top of the abstraction hierarchy. These applications are implemented in **High-Mid Level Languages**, such as C/C++, Python, Java, etc. These programming languages have their own hierarchy, but for the purpose of this thesis, it is sufficient to understand that these programming languages are then compiled into **Assembly Language**. Assembly language closely follows the execution of instructions on the **processor**, and is defined by an **ISA**. An ISA can be thought of as the contract between software programmers and processor engineers, agreeing what instructions the processor is able to implement. This assembly code is finally loaded into the processor's instruction memory and executed.

`chapters/chapter1/img/computerHierarchy.png`

Figure 1.2: Simple Visualization of Computer Implementation Hierarchy

RISC-V

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

Move the Hardware and RISC-V subsubsections to proposed solution?

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

This thesis aims to design hardware that executes motion planning algorithms faster than those same algorithms can execute on generic hardware. Chatper 2 introduces motion planning and details the process of implementing and analysing RRT to identify computational hot-spots in the algorithm and thus identify the biggest opportunities for hardware optimization.
Section 2.1 provides an introduction to Motion Planning Algorithms in general. Section 2.2 outlines the RRT algorithm, and describes the implementation of RRT for this project. Finally, Section 2.3 outlines a method for performance analysis of RRT and the results of such analysis.

Write a better introduction that more accurately defines sections and makes the POINT of this chapter clear.

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

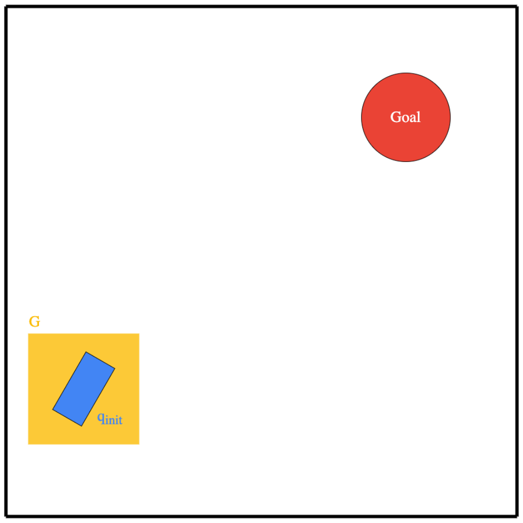
Put simply, RRT builds a tree (referred to as a graph) of possible configurations, connected by edges, for a robot of some physical description. It does so by randomly sampling the configuration space and adding configurations to the graph. From this graph, a path from the initial configuration to some 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

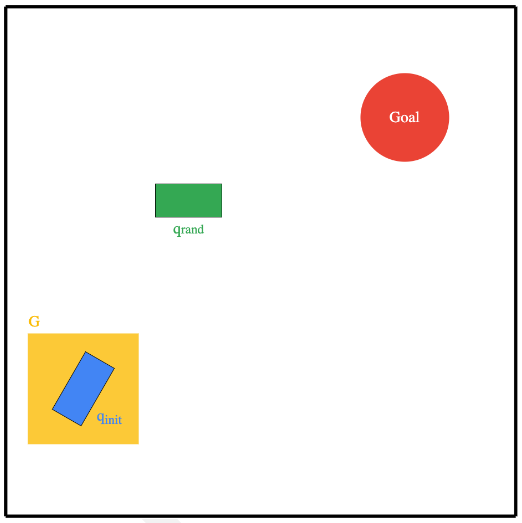
```
Inputs:  Initial configuration $q_{init}$,
         Number of nodes in graph $K$,
         Incremental Distance $\Delta q$
Output:  RRT Graph $G$ with $K$ configurations $[q]$ & edges $[e]$
$G$.init();
for $k = 1$ to $K$ do
    $q_{rand} \leftarrow$ randomConfiguration();
    $q_{near} \leftarrow$ nearestVertex($q_{rand}$, $G$);
    $q_{new} \leftarrow$ newVertex($q_{near}$, $q_{rand}$, $\Delta q$);
    $G$.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

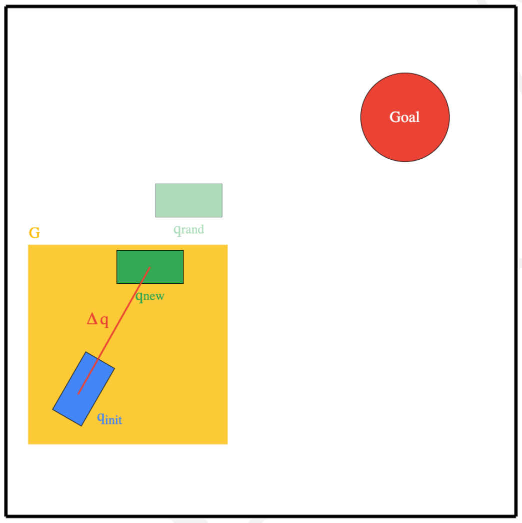
2.1c). The distance between q_{near} and q_{rand} is calculated. If this distance is less than Δq, $q_{new} = q_{rand}$. If not, q_{new} is selected, typically by moving by Δq from q_{near} towards q_{rand} (2.1c). q_{new} is then added to G. This is repeated for K configurations.



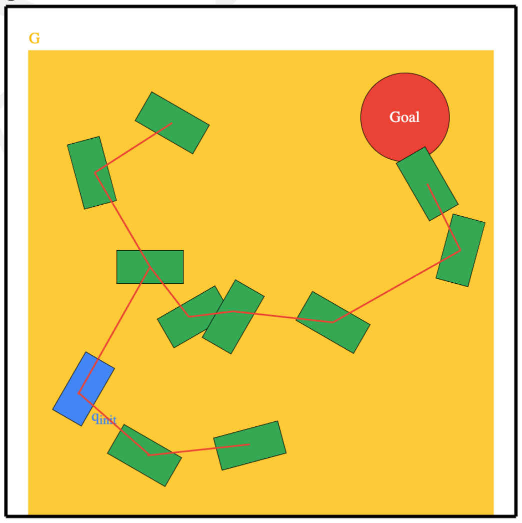
(a) Graph G contains only q_{init}



(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 $\Delta q$,
         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($q_{near}$, $q_{rand}$, $\Delta q$);
    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 2.3.1, 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

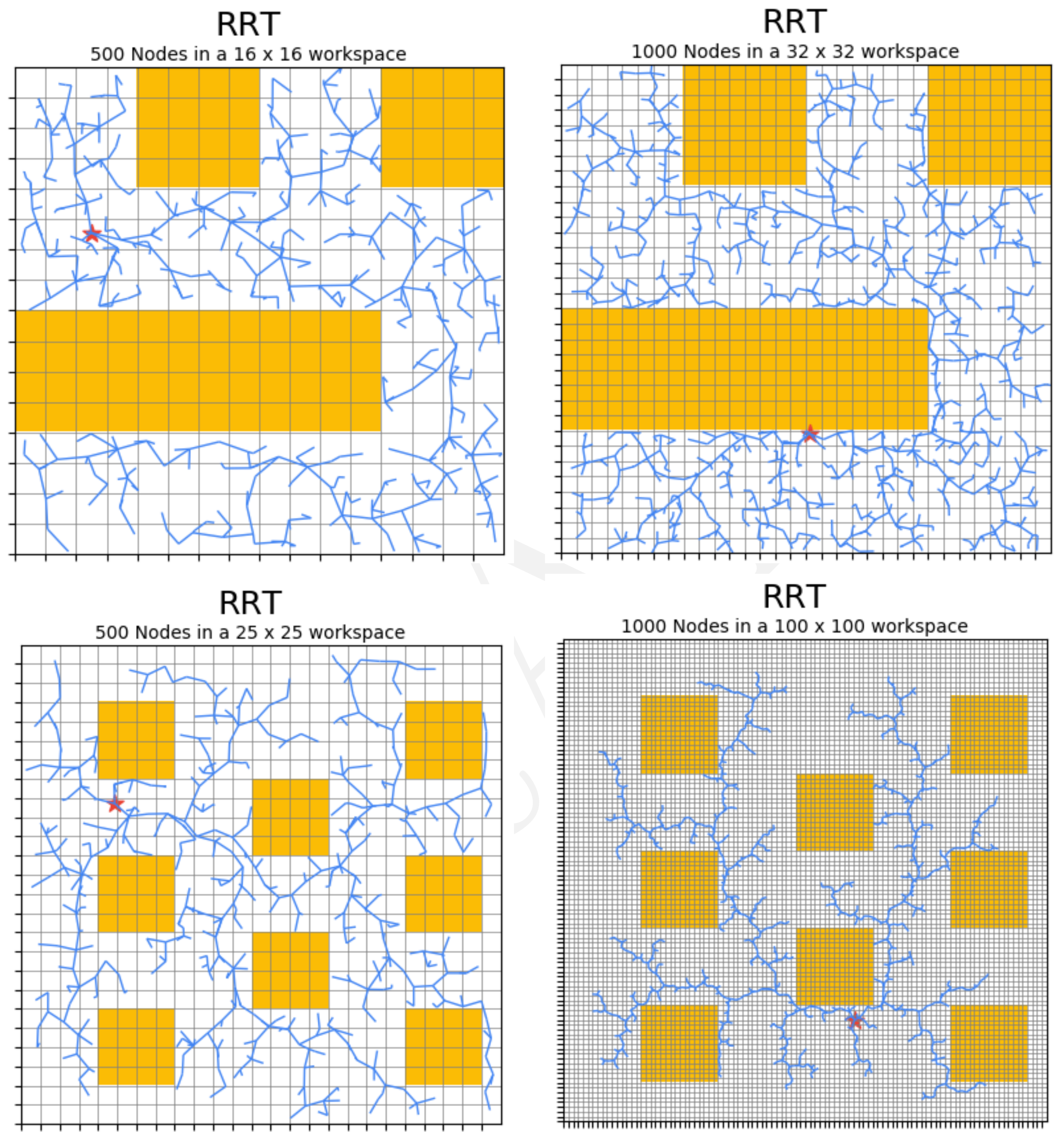


Figure 2.2: 2D RRT Implementation shown by GUI

Implementation in 3D

Describe implementation in 3D

Figure 2.3: 3D RRT Implementation shown by GUI

2.3 Performance Analysis

Brief introduction outlining purpose of performance analysis

2.3.1 Methodology

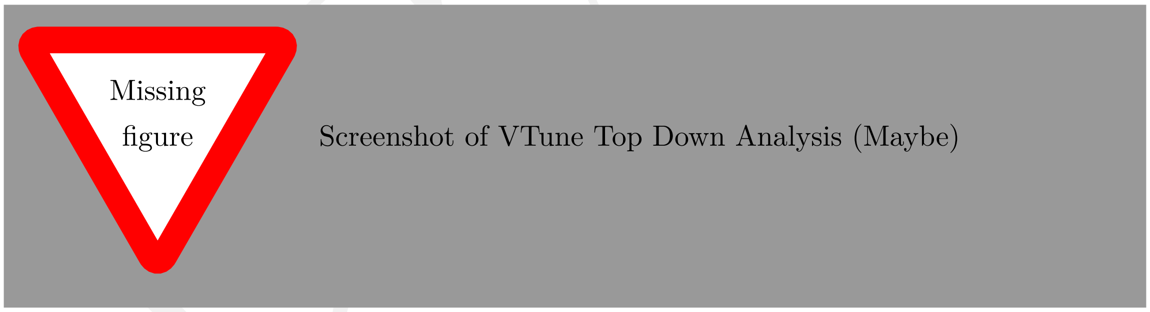
To restate, the aim of this thesis is to design a computer processor with reduced execution time of motion planning algorithms, such as RRT. As such, it is important to understand the elements of the algorithm that have the highest percentage of CPU execution time. To determine this, it was necessary 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.

Outline of method of analysis. Something better than the above

VTune Profiler

VTune Profiler 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



Screenshot of VTune Top Down Analysis (Maybe)

Figure 2.4: VTune Amplifier TopDown Analysis Example

Internal Timing

The limitation of VTune Profiler is that it can only profile software running on Intel processors, which implement the x86-64 ISA. As such, when the time comes to analyse performance of the software running on a RISC-V processor, another method will be required. A simple and effective way of measuring execution performance is to insert timing functionality into the software itself.

Provide or link to appendix of explanation of internal timing

Comparison

Before proceeding to use either of these methods to profile the software implementation of RRT, it was important to verify that the two methods yielded similar results for the same program. Table 2.2 summarizes the results of analysis of a simple C executable. The program calls 5 functions, $\{A, B, C, D, E\}$, each a simple iteration in which a integer is incremented. Since the Internal Timing method returned similar results to the (trusted) VTune Profiler, it was considered to be a reliable method. While it was encouraging to see both methods returned similar results for absolute execution time, the more important metric was the similarity in percentage of total execution time.

function	Vtune Profiler		Internal Timing	
	time (s)	time (% total)	time (s)	time (% total)
A	0.488	57.4%	0.497	57.6%
B	0.2	23.5%	0.198	23.1%
C	0.102	12.0%	0.099	11.5%
D	0.048	5.7%	0.049	5.6%
E	0.012	1.4%	0.019	2.2%

Table 2.2: Comparison of Timing Methods

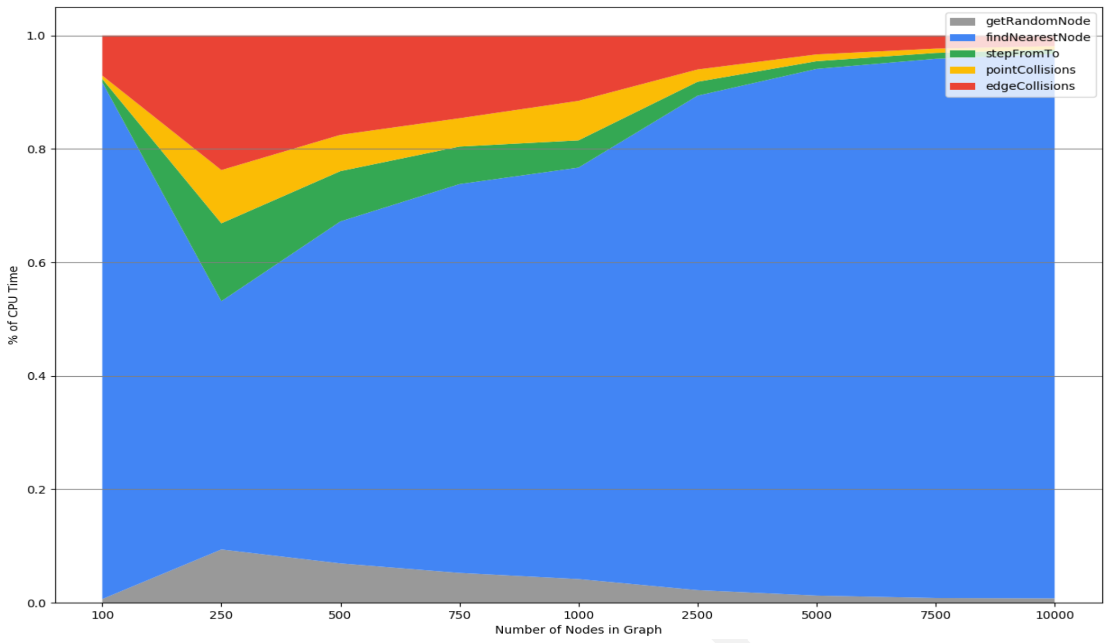
Experimental Design

In profiling RRT in software, the goal was to find the critical task across different values of K and sizes of configuration space. Multiple tests were run, varying these two constraints, to find this critical function. The results of this analysis can be found in Section 2.3.2.

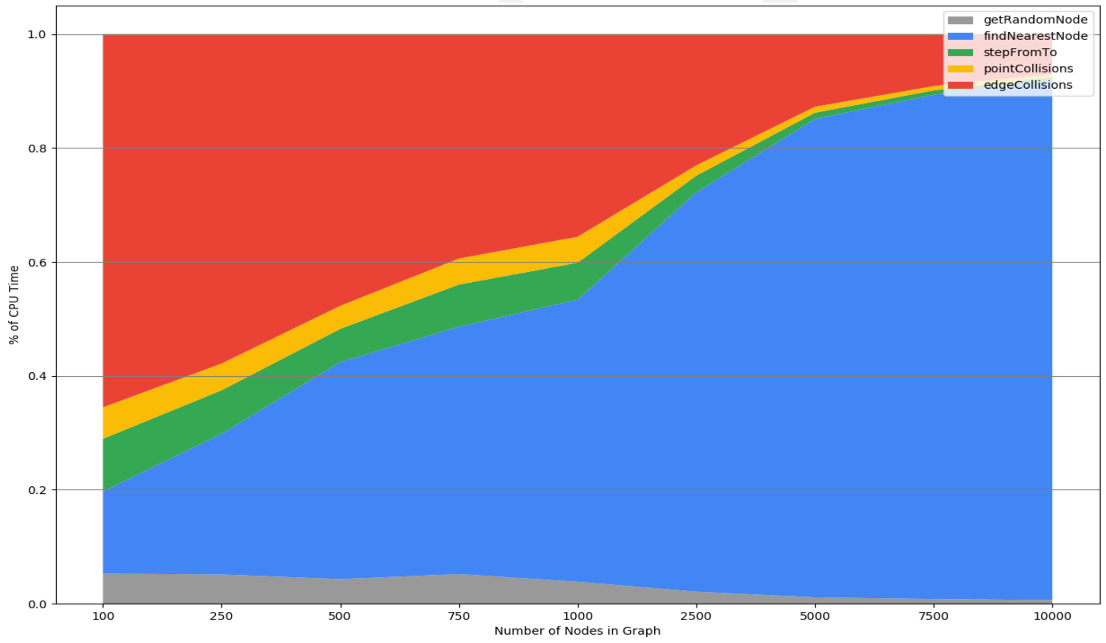
2.3.2 Results

Figure 2.5 shows the profile of functions within RRT, for $100 \leq K \leq 10000$, and cubic configuration spaces with dimensions $\{4, 8, 16, 32\}$. Each subfigure shows a similar profile, with the % of CPU Execution Time taken by findNearestNode increasing with K. This is to be expected. . However, it is also seen that edgeCollisions increases with larger configuration spaces, taking up the overwhelming majority of execution time for a 32x32x32 configuration space.

Explanation of time complexity

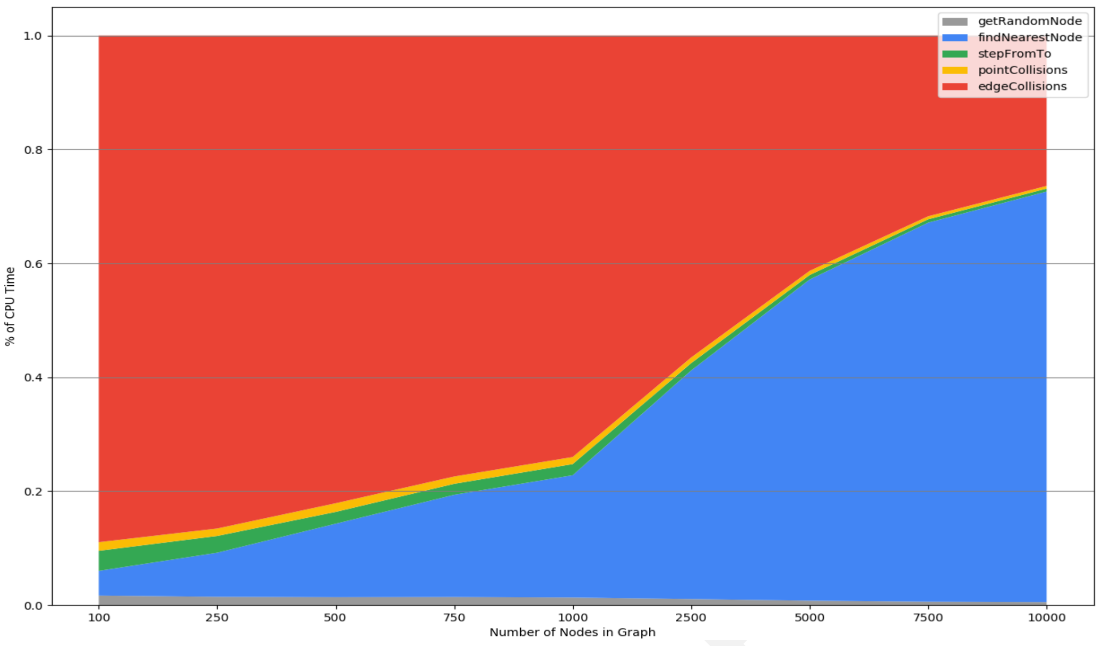


(a) 4x4x4 Configuration Space

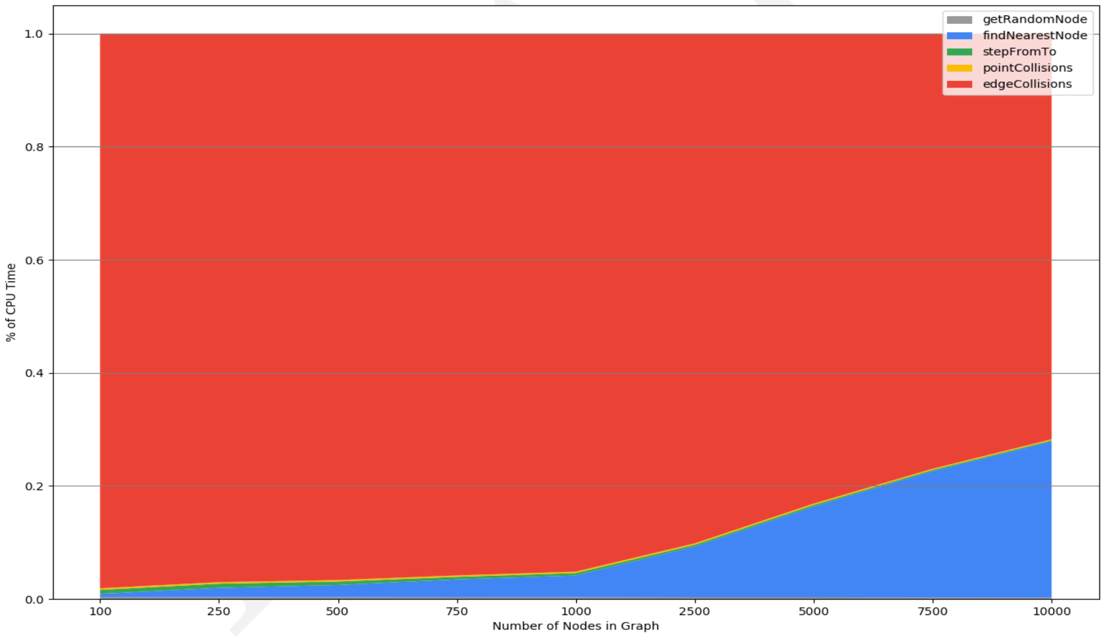


(b) 8x8x8 Configuration Space

Figure 2.5: RRT Functions as a % of Total CPU Exectution Time



(c) 16x16x16 Configuration Space



(d) 32x32x32 Configuration Space

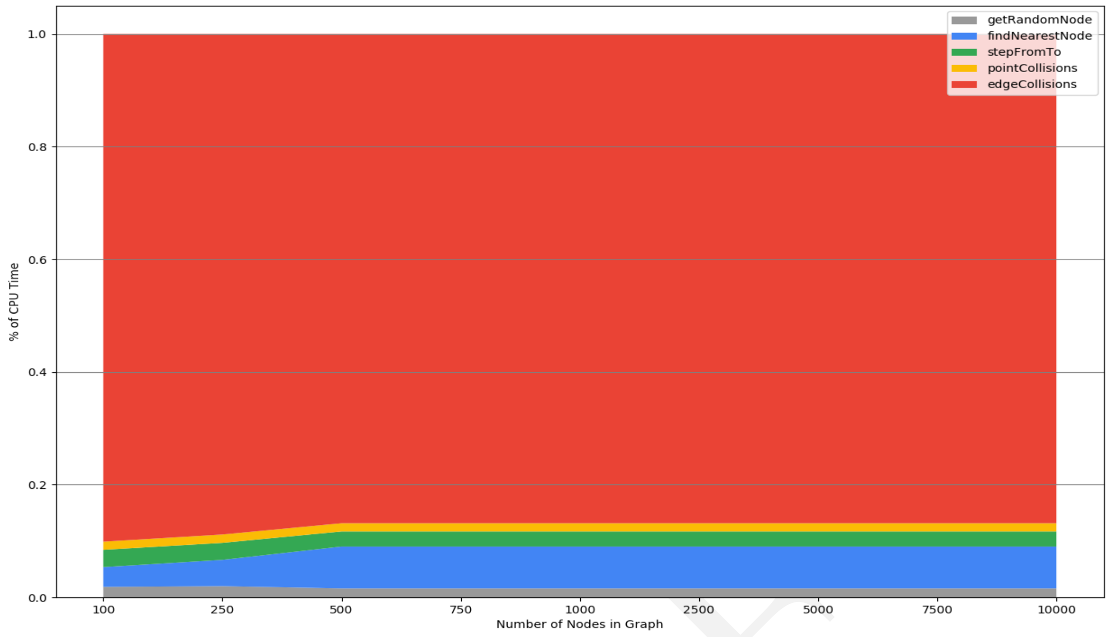
Figure 2.5: RRT Functions as a % of Total CPU Exectution Time (cont.)

Change Y axis to % and increase text size

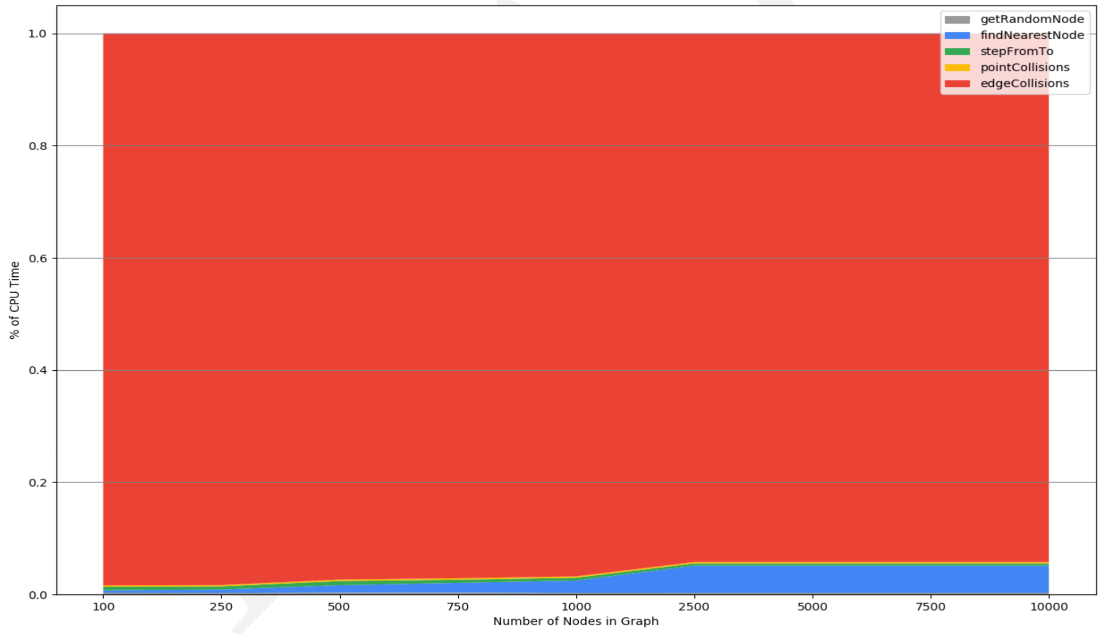
Furthermore, the computational load of findNearestNode can be reduced through a variety of software optimizations. A simple one used here to demonstrate that fact is storing nodes in seperate “buckets,” sorted by their x value. By using only two buckets, the execution time of findNearestNode fell drastically. Figure 2.6b shows edge collision detection accounting for over 95% of execution time for $100 \leq K \leq 10000$. This is consistent with the profiling results of RRT in prior work[8].

Conclusion

From the above data, it was identified that, as prior work suggested, edge collision detection shows the greatest promise for potential speedup through specialized hardware. The next chapter details the process of designing and building this hardware.



(a) 16x16x16 Configuration Space



(b) 32x32x32 Configuration Space

Figure 2.6: RRT Functions Exectution Time, with Bucket Optimization

Add simulations to determine correct K

Chapter 3

Motion Planning in Hardware

3.1 Background to Hardware Optimization

Brief introduction to the concept of hardware optimization and examples of work in the field

3.2 HoneyBee

The Honey Bee has long been renowned for its tireless work ethic. But people rarely give the Honey Bee credit for its remarkable navigation and collision avoidance strategies during flight. As such, it is quite appropriate that this functional unit, designed to work tirelessly, rapidly and efficiently to execute collision detection computations, is named HoneyBee.

3.2.1 Design

It was demonstrated in Section 2.3.2 that the critical function of RRT was edge collision detection.

3.2.2 Build

High Level Synthesis

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`.

[8] J. Bialkowski, S. Karaman, and E. Frazzoli, “Massively parallelizing the RRT and the RRT,” in *IEEE International Conference on Intelligent Robots and Systems*, pp. 3513–3518, 2011.

Appendices

Appendix A

Project Repository

TODO: Provide information about this project's repository

Todo list

Use glossary package iv
Use better acronym package that includes plurals iv
cite 1
Figure: Some sort of line/bar graph showing the increasing use of Autonomous robots over time. Need to find 2
TODO: More of an introduction to motion planning. 2
TODO: Introduction to RISC-V and its merits in this problem 3
Move the Hardware and RISC-V subsubsections to proposed solution? 3
Revise problem statement 4
TODO: Revise End User 4
TODO: Summary of prior work 4
Proposed Solution 4
Project Specifications 4
Project Structure/Timeline 5
Write a better introduction that more accurately defines sections and makes the POINT of this chapter clear. 6
TODO: Background on Motion Planning Algorithms. 6
Better RRT Implementation introductory sentence 10
Change this based on whether time does permit 10
Improve this table 10
More detail 11
Describe implementation in 3D 12
Figure: Simulations of 3D RRT Implementations 12
Brief introduction outlining purpose of performance analysis 13
Outline of method of analysis. Something better than the above 13
Rewrite the above 13
Figure: Screenshot of VTune Top Down Analysis (Maybe) 13
Provide or link to appendix of explanation of internal timing 14
Explanation of time complexity 14
Change Y axis to % and increase text size 17
Add simulations to determine correct K 19

Brief introduction to the concept of hardware optimization and examples of work in the field 20
TODO: Provide information about this project's repository 26