CPU SCHEDULING IN ROBOTICS & AR/VR APPLICATIONS

BY

ADITI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Advisers:

Associate Professor Philip Brighten Godfrey
Assistant Professor Radhika Mittal

ABSTRACT

Robot and AR/VR systems have to take highly responsive real-time actions, driven by complex decisions involving a pipeline of sensing, perception, planning, and reaction tasks. Given constrained resources, this leads to a difficult scheduling problem. In practice – system designers manually tune params for their specific hardware and application, while real-time scheduling approaches assume static periodic schedules – both of which result in suboptimal application performance especially when both the environment and the hardware can change.

In this work, we highlight the emerging need for automated resource optimization at runtime in sense - react systems. As a step towards this goal, we identify various unique challenges in this area, especially understanding the key scheduling requirements for such systems. We propose a preliminary framework and a novel scheduling policy that enables efficient and dynamic optimization of application-specific performance goals. In experiments with a prototype implemented in the ROS and ILLIXR platforms, we show that our approach improves application performance, for example, 15x better performance for face tracking robot and 7x better collision avoidance for a navigation robot. We believe this work will lead to systems that are substantially easier to develop and fulfill their tasks measureably better.

To my parents, for their constant love and support.

ACKNOWLEDGMENTS

I have received a great deal of support and assistance throughout my Masters degree, for which I am very thankful.

First and foremost, I would like to express my sincere gratitude to my research advisors, Dr. Philip Brighten Godfrey and Dr. Radhika Mittal. Their insights, encouragement and feedback have helped me grow as a researcher.

Next, I would like to acknowledge my collaborators, Professor Sarita Adve, Muhammad Huzaifa, and especially Samuel Grayson - I really enjoyed working with you and got to learn a lot during all the debugging sessions :')

I would also like to thank my family, for always being there for me. Last but not least, I could not have completed this dissertation without the support of my friends, Kapil Vaidya, Ashish Kashinath and Pulkit Katdare, who provided happy distractions during the stressful times of my research.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 3
2.1 Overview of Robot and XR systems 3
2.2 Deployment Platforms 5
2.3 Development Frameworks 5

CHAPTER 3 UNIQUE CHALLENGES & OPPORTUNITIES 7
3.1 Challenges 7
3.2 Opportunities 13

CHAPTER 4 RELATED WORK 14
4.1 Real Time Scheduling 14
4.2 Stream Processing 15
4.3 Offline Profiling 16

CHAPTER 5 RESOURCE MANAGER DESIGN 17
5.1 Problem Formulation 17
5.2 Stage I: Core Allocation 18
5.3 Stage II: Per Core & Per Subchain Scheduling 21
5.4 Handling Dynamicity 23

CHAPTER 6 IMPLEMENTATION 24

CHAPTER 7 EVALUATION 26
7.1 Face Tracking Robot 26
7.2 Robot Navigation 28
7.3 Virtual Reality 31

CHAPTER 8 CONCLUSION & FUTURE WORK 33

REFERENCES 34

CHAPTER 1: INTRODUCTION

Sense - react systems such as robotics and AR/VR are becoming ever more pervasive. While a variety of robots now assist us with various mundane, high precision, or dangerous tasks [1, 2, 3, 4, 5, 6, 7, 8, 9], XR has affected the way we teach, practice medicine [10, 11], etc, and is envisioned to be the next interface for compute [12, 13]. Both robots and XR systems typically feature a sense-perceive-plan-react pipeline (Figure 1.1) which ranges from a single linear chain of components in the simplest applications to a multi-chain directed acyclic graph (DAG) in more complex ones. Most such systems run on devices with limited compute, e.g. a mini PC [14, 15] or Raspberry Pi [16, 17] or In this work, we motivate the need for automated and dynamic resource management in such compute platforms, identify the key challenges, and develop a preliminary scheduling framework to dynamically manage on-device CPU resources for sense - react systems at runtime.

Resource management for robotics and AR/VR requires co-optimizing multiple dimensions – how much resources must be allocated to each component, how often each component gets triggered, and the order in which they run. This results in a difficult scheduling problem, where the appropriate choice depends on the amount of available resources, the resource usage of each component, and the performance requirements of the application. The resource manager must also take into account dynamic variations in resource usage (e.g. due to accumulation of more information about the environment, or a change in scene) and in resource availability (e.g. due to battery constraints). The fact that the impact of scheduling decisions on the performance of a sense react system is deeply tied to semantics of the specific application, further add to the difficulty of resource management.

Despite the rich literature in system scheduling, past work fall short of addressing these requirements and challenges in sense - react systems. They either optimize a subset of the dimensions listed above (e.g. just the triggering frequency [18], or the execution ordering [19, 20, 21]), or consider local (per-component) adaptations that do not take system-wide requirements into account [22, 23, 24, 25, 26, 27, 28], or rely on static configurations learnt offline that cannot adapt to dynamic variations [29, 30, 31, 32, 33]. To the best of our knowledge, none of the related works discuss the correlation between scheduling decisions and applications' performance.

Thus resource management in sense - react systems remains a challenge in which developers are provided little help. While frameworks that assist in development of applications exist (e.g. ROS [34], ILLIXR [12], OpenXR [35], they leave resource management decisions entirely up to the application developers in the form of manually configurable parameters.

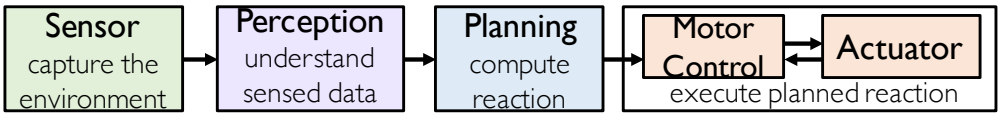

Figure 1.1: Basic pipeline in sense - react systems

Application developers often react to poor performance with desperate attempts at fine-tuning the scheduling decisions for different components, scaling up the resources, and other such ad hoc strategies. The engineering effort involved in hand-tuning system performance steals from the time that could have been better spent on developing more useful higher-level applications, and also increases the barrier of entry into the field of robotics and AR/VR.

We therefore highlight the key scheduling challenges and requirements, and design Jenga (§5), an automated resource manager to dynamically manage the compute resources for a sense react application. Jenga uses a hierarchical approach to solve the scheduling problem, and has support for handling various kinds of application requirements. It can be plugged into existing robotics and AR/VR frameworks (e.g. ROS, ILLXIR respectively), and provided as an optional service to the applications using the framework. Through our preliminary case studies (§7), involving 3 applications in both domains (object tracking robot, exploration robot and Virtual Reality), we show how Jenga can achieve better application performance than the default (hand-tuned) configurations, and demonstrate the significance of meeting the scheduling requirements, specifically handling dynamicity and heterogeneity in the DAG. To the best of our knowledge, this is the first work to evaluate the impact of system scheduling on how well a sense - react system fulfills its tasks.

Our work thus motivates the need for automated resource management at runtime in order to improve a system's performance and ease application development. While, as a first step, we focus on managing CPU resources on the edge device for robots / XR, our work opens up several interesting directions for future research (§8), which include generalizing to systems with heterogeneous compute resources (e.g. CPU, GPU), simultaneously managing memory and network constraints, and to other sense-react applications.

CHAPTER 2: BACKGROUND

2.1 OVERVIEW OF ROBOT AND XR SYSTEMS

Most robotics and AR/VR applications can be characterized by components connected in a DAG, where each node represents a computation task (or component) and each edge represents flow of data between tasks. Different applications vary greatly in their complexity, based on the high-level task to be accomplished. In this section, we take a look at 3 such applications, and also discuss the task - specific goals for each of them.

2.1.1 Face Tracking Robot

A simple face-tracking application [36, 37] is comprised of nodes arranged in linear pipeline (Figure 2.1) – a camera (C) to capture and pre-process images, a perception node (OD) to find the face in the image, a planning node (VP) to compute the velocity at which the camera must rotate to track the face, and the camera motor that rotates accordingly.
Performance Goals. The main goal of this robot is to always keep the object in its camera frame, or equivalently, to never lose track of the object.

2.1.2 Robot 2D Navigation

The robot's task in this application is to explore an unknown area [38]. Figure 2.2 represents how the various components within the application are connected to form a DAG. It uses two sensors – a laser scanner and an odometer, which reports speed and location information at a high rate to all other nodes. The local cost mapper (LC) uses the laser scans to update its knowledge of the robot's immediate vicinity. The global map and localize node (GML) performs two tasks: (a) Correcting the robot's location and (b) pre-processing the laser scans before sending them to the global mapper (GM) that maintains a global map of all the areas that the robot has explored so far. The global planner (GP) computes the global trajectory of the robot, i.e. which area to explore next, based on the current position of the robot and GM's global map. The Navigation Command (NC) node, based on the robot's position and the current trajectory (GP's output), decides in what direction should the robot move. Finally, the local planner (LP) uses the local cost map and the navigation command to output the robot's velocity, which is actuated by the mobile base.
Performance Goals. For the navigation application in a dynamic environment, avoiding

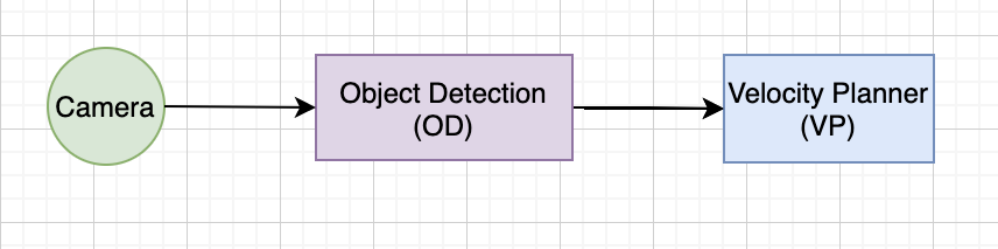

Figure 2.1: DAG representing the face tracking robot application

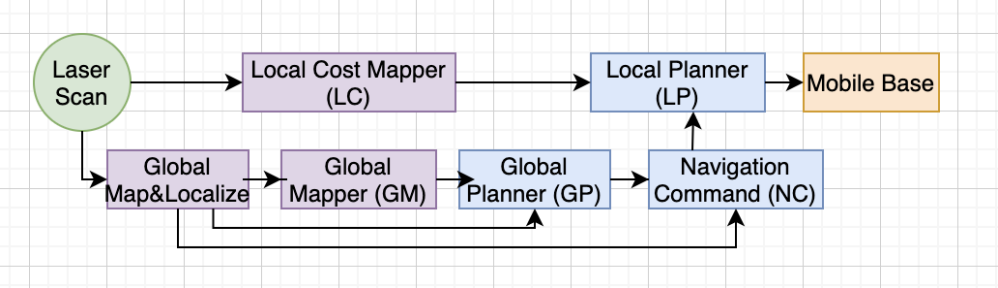

Figure 2.2: DAG representing the robot navigation application

obstacles is critical, and that is the first performance goal that we consider. A second goal is to explore the entire area quickly, i.e. to have a high rate of exploration.

2.1.3 Virtual Reality

The VR application's task is to provide a smooth visualization of the virtual world, based on the user's head movement. Figure 2.3 represents the various components within the application. It uses two sensors – an IMU, which measures acceleration of the user's head, and a camera, which takes pictures of the surrounding environment. The SLAM component fuses the camera and IMU sensors to produce an estimate of the user's pose. The integrator (Int) integrates incoming IMU samples from the IMU to obtain a relative pose, and combines it with SLAM's output to get a better pose estimate. The Render task (R) uses this pose along with Int outputs to computes an image (aka 'frame') of the virtual world, which would be from the perspective of the pose estimate. The Timewarp component takes in this rendered frame and processed IMU values from Int, and updates the frame according to the user's pose based on the IMU data from Int. Lastly, the timewarp's output is sent to the display driver so as to display the frame on a screen mounted to the user's headset.

Performance Goals. To provide a smooth virtual visualization, the application should estimate the user's pose correctly [so that the rendered image is in the correct perspective], generating the right image and updating the virtual world state with as less delay as possible.

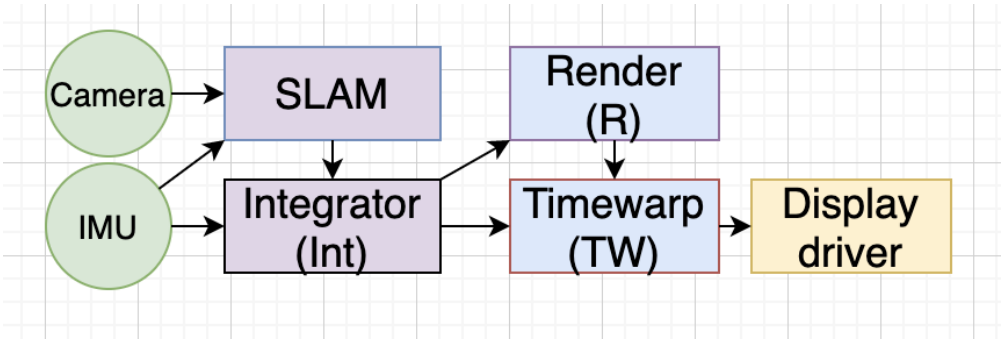

Figure 2.3: DAG representing the VR application in iLLIXR

For example, in a bowling simulation, the application should quickly detect if a player crosses the fault line so as to update the scores accordingly.

2.2 DEPLOYMENT PLATFORMS

In many low cost robots, sensors and actuators are attached to a single on-board computer which runs all software components in the DAG, often using CPU as the only compute resource [14, 17, 39]. GPUs and other accelerators can be attached at the cost of higher expense and battery usage. Note that this means that there would be contention among the different components for usage of the CPU.

In larger robot systems, the nodes may be split across multiple on-board machines [40**?**], or some nodes may be offloaded to edge or cloud servers [41, 42, 43, 44].

There are some VR headsets which offload the computation to an attached server [45]. But more commonly, many AR/VR applications are run with the support of an embedded system [46, 47, 48], on stand-alone devices such as Oculus Go & Quest [49, 50]. The embedded platforms provide on-board compute in the form of CPUs, GPUs and accelerators such as DSPs.

In this work, we focus on systems using a single on-board computer, with CPU as the only contended compute resource.

2.3 DEVELOPMENT FRAMEWORKS

It is common to use a software *framework* (e.g. [12, 34, 35, 51, 52, 53, 54, 55]) for developing sense - react applications for both robotics and AR/VR. The modularity of such frameworks allow developers to easily re-use existing software packages, swapping specific nodes with newer ones, as needed.

Robot Operating System (ROS) [34, 56] is by far the most popular framework for robotics applications, with more than 300K estimated users [57]. It is widely used for developing research prototypes, with industry usage also growing rapidly [2, 58, 59, 60, 61].

Illinois Extended Reality testbed (ILLIXR) [12] is a new development framework for AR/VR applications, which provides the implementation for various widely used components, along with an extensible communication interface and runtime. It is compliant with OpenXR [35], which is a standard for a standard for XR device runtimes to interface with the applications that run on them.

Both the frameworks allow developers to program each component of an application individually, and provide APIs for communication (i.e. flow of data) among those components. There are two broad types of communication supported by both ROS and ILLIXR, listed below. Let us consider there is an edge from component A to B in the DAG of an application [i.e. A's outputs are used by B]. Then,

- *Synchronous.* B is synchronous w.r.t. A (or equivalently this edge is synchronous) if it waits for A's outputs and executes exactly once for each output of A. This is similar to the publish subscribe or event driven messaging [24, 26] and can be modelled as a message queue b/w AB with insertions by A and deletions by B. One can set the maximum queue length as a parameter in ROS.

- *Asynchronous.* B is asynchronous w.r.t. A (or this edge is asynchronous) if it uses the latest output of A every time it executes.

Note that each component's execution can either be triggered [1] by a synchronous edge or have a periodic timer based on a tunable frequency parameter.

ROS supports using TCP/UDP for communication among different components if they are in different processes, and using shared memory if they are different nodelets within the same process. ILLIXR supports shared memory communication since all components are implemented as different plugins (threads) in the same process.

[1] The node executes once for every trigger it receives.

CHAPTER 3: UNIQUE CHALLENGES & OPPORTUNITIES

The problem of CPU scheduling for DAGs representing sense - react systems is hard, and poses many challenges unique to such systems. In this chapter, we will discuss these challenges, along with certain opportunities that one can leverage in such systems.

3.1 CHALLENGES

3.1.1 Resource Management Decisions

Management of CPU resources for a DAG - like application involves co-optimizing several dimensions of scheduling. Namely, we need to take the following decisions :

- Core Allocation. Given a multi core platform, one needs to decide which components are allowed to use which cores. We should also account for heterogeneity among cores, and hyper threading within each core.
- Execution Frequency. For each component, one needs to decide the rate at which it should execute, i.e. use inputs from upstream nodes to produce a new output.
- Degree of Parallelism. There can be components which support multi - threading, e.g. the computation within the GM component in the navigation DAG can be parallelized. For such components, one also needs to decide the level of parallelism, or equivalently, the number of threads it is allowed to make.
- Temporal CPU Allocation. For a core to which multiple components are assigned, one needs to decide the scheduling order among those components. e.g. In a priority based scheduling policy, the relative priority numbers are a parameter, that affect which component gets to use the cpu core at any given time.

It is a hard scheduling problem to solve for the optimal set of scheduling decisions. We have a simple proof of NP-hardness for a subset of this problem in [62]. The optimal solution would depend on the available resources, the compute requirement of different components, as well as the performance - related requirements of the application itself.

3.1.2 Dynamicity

The scheduling problem is exacerbated by the fact that there can be dynamicity in resource usage & availability, as well as the environment.

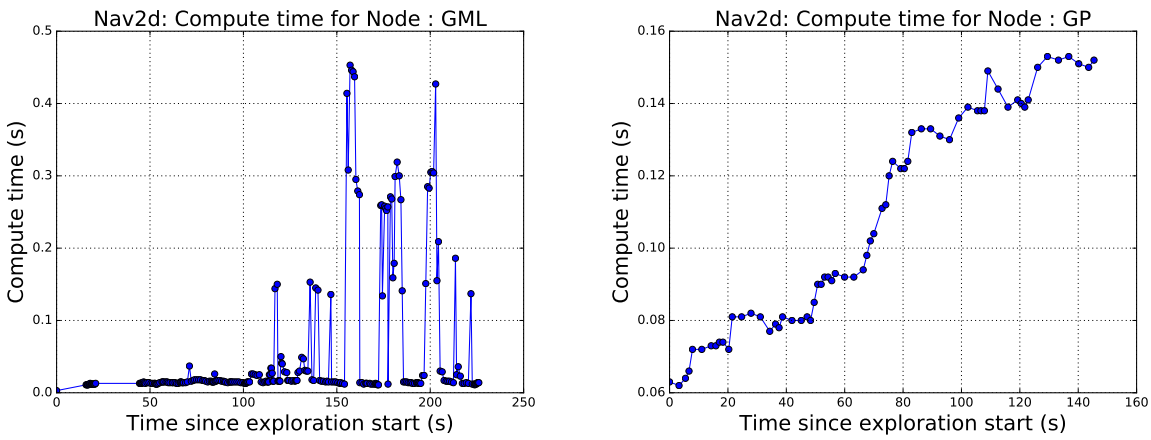

Figure 3.1: Variation in the CPU resource usage of various components (GML, GP) of the navigation application, for exploration in a particular environment.

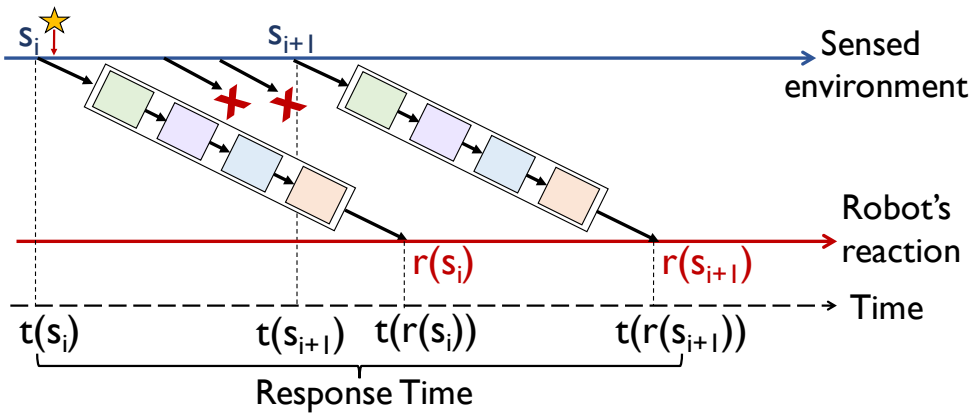

Figure 3.2: Response Time for a chain - the yellow star denotes a change in the environment, and RT denotes the worst case time for the chain to capture this change and react to it.

Resource Usage. The compute resource requirement of a component can vary over time, or based on its inputs. For example, as the navigating robot covers more area, the compute time taken by the component GM to generate the global map increases, as it has to accumulate all the information collected since the start of exploration. As the size of GM's global map increases, the compute time to decide the robot's trajectory by GP also grows. Lastly, the node GML can perform loop closure [63] while correcting the robot's location, which is very expensive, and causes spikes in GML's compute time graph. Figure 3.1 shows the above trends. In the ILLIXR application, the SLAM node similarly has high variability in resource usage.

Resource Availability. The amount of resources available to the application can vary with

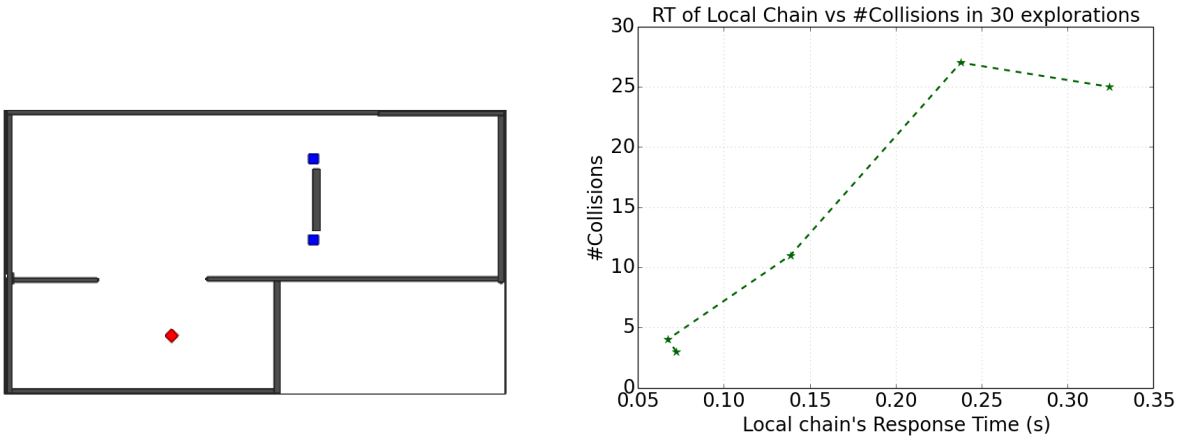

Figure 3.3: (Left) A small map with 2 obstacles (blue), which start moving towards the opposite wall when the robot (red) arrives near, so as to block its path. (Right) Number of times the robot collides with an obstacle, when navigation is run 30 times (in simulation) on the map on the left, with local chain's RT on the x-axis. [All other nodes were running on separate cores at default frequencies]

time, or across different hardware platforms. For instance, for any embedded platform/robot, as time goes on, one might reduce the number of CPU cores, or reduce the frequency of the cores, so as to save power/battery life.

Environment. Different types of environments can also vary the application's requirements, and consequently the scheduling decisions. For example, a navigating robot might need to update the local area map more quickly in an environment with dynamic obstacles, as compared to a static environment. In the case of VR, a dynamic game such as PUBG will have much more stringent requirements than a simple scenario like watching the sky [which has a static background].

All in all, because of all these kinds of variations, it is important that the resource manager is aware of them and has the ability to adapt the optimal decisions accordingly.

3.1.3 Heterogeneous Low – level Requirements

For designing a solution, it is important to analyse the effects of the scheduling decisions on the performance metrics. Since it requires considerable effort to experiment with various possible scheduling decisions via simulations/real systems to compute the corresponding application performance, it is easier and more intuitive to consider low-level metrics to bridge the gap. In other words, if we understand the correlation between low level metrics and the application's performance, then the scheduling problem can be simplified into meeting

requirements on the low level metrics.

Let us first discuss some commonly used low level metrics in DAG applications. We define each unique path from an input node (with no incoming edges) to an output node (with no outgoing edges) as a *chain*. As can be seen in Figures 2.1, 2.2 and 2.3, the object tracking application, navigation and VR have 1, 4 and 6 chains respectively. We now list some low level metrics :

Chain Latency. We can define the latency along each chain in the DAG to represent the time taken for for an input at the source (i.e. a sensor) to be processed along chain and generate an output at the sink (last component in the chain).

Component Throughput. This represents the rate at which a given component produces new outputs.

Chain Throughput. We can define the throughput along a chain as the rate at which inputs from the chain's source node (a sensor) are processed to generate new outputs at the sink.

Chain Response Time. We define response time (RT) along each chain as, the worst-case time taken for a change to occur in the environment, for it to be captured by the source (sensor) node, and the corresponding reaction to be produced at the sink of the chain. Figure 3.2 denotes this metric. Intuitively, RT is the latency along a chain plus the time gap between two outputs (it thus combines both latency and throughput). This metric shares similarity with "age of information", a freshness metric defined in the context of network communication [64].

We can now analyse some intuitive correlations between performance and low level metrics. For the face tracking application involving a single chain, the ability of the system to track the object strongly depends on how long it takes the application to process and react to a new camera image (which contains information about the object's latest position). Hence, we expect to see a correlation between the response time (RT) of the chain and the robot's performance, and optimizing for RT would be the low-level requirement that a scheduler can focus on.

For the navigation application, how well a robot is able to avoid dynamic obstacles, depends to an important degree on how quickly the LC and LP nodes process new scans to update the map of the immediate environment and calculate the robot's velocity accordingly, i.e. the throughput and latency along the *local chain* 'scans $\rightarrow$ LC $\rightarrow$ LP $\rightarrow$ base'. We also observed this emipirically, in a simple setting for this application wherein the robot was able to avoid collisions better when it had better response time for this chain (wile keeping all other scheduling decisions fixed), as shown in Figure 3.3.

In the VR application, it is important to have low latency along the IMU - Int - TW - Display chain so as to avoid nausea in the VR user [65, 66]. This again represents a direct

correlation in application performance and low level metrics, and the scheduler design can take this into account.

But apart from these intuitive insights, it turns out that such correlations are deeply tied to application semantics, and require extensive inputs from domain experts / developers. We now discuss various such insights that highlight both the difficulty in getting the low level requirements, as well as challenges for the scheduling problem itself.

- The node GML buffers laser scans, so as not to lose out information contained in the scans, about areas that have been visited, which is in contrast to nodes such as NC, GP which always use the latest position estimate, i.e. the latest odometry, and LC similarly uses the most recent (latest) laser scan to update its map so as to reduce the delay in sensing & reacting to obstacles. This translates to different low level metrics being important for different components.

- The GML has a bimodal nature - for each laser scan, it either does a quick check & discards the scan if it does not have any new information, or processes the scan [which involves updating the pose correction, possibly performing loop closure, and sending the scan to GM]. Lastly, for each scan [whether processed or discarded], it sends out the pose correction. Hence it is important to not batch the processing at GML, since both GP, NC combine this correction with the position (odometry) information with the same timestamp, and they will end up using old odometry if GML's outputs are delayed. Each of these characteristics differentiate this component from others.

- The node GM runs only if there's at least one new scan (coming from GML). Hence, the effective frequency of such a node will highly depend on the GML's logic of discarding laser scans. We found that GML node has a few configuration parameters (travel distance [67], travel rotation [68]), which affect how many scans does GML discard (if the robot has moved less than travel distance and rotated less than travel rotation, GML discards the scan), and consequently affect the throughput of both GML and GM.

- The node GP has a smart logic of when to execute in the codebase, it runs only if (a) its been too long since it last executed, (b) the robot has followed almost the whole of the previous trajectory planned by NP. While trying to execute the planner at a rate higher than the maximum frequency set in the code (1Hz), we observed a lot of thrashing, i.e. the GP will output a trajectory to a certain location (goal 1) for a while, and then update the trajectory to move towards another location (goal 2), but then again change the path towards goal 1. The time spent by the robot trying to reach

goal 2 might have been wasted in such a scenario, and can lead to worse application performance. This results in a counter intuitive effect of improving a low level metric (running GP faster) not necessarily leading to better performance.

- In the case of VR, one needs to send a new frame (i.e. TW's output) to the display driver at a *fixed* frequency (called vsync) which is a function of the display's hardware. The scheduler should allow adding such constraints on the throughputs of components.

To summarize, the various kinds of heterogeneity listed above make the job of the scheduler design complicated - it needs to be general enough so as to take into account all such requirements, while solving for the best scheduling decisions.

3.1.4 Inferring Low – level Requirements

As discussed above, the requirements and constraints on low level metrics are deeply tied to the application semantics, and can be very complex and heterogeneous. For instance, there can be applications where executing the global planner at a higher rate always helps performance. Not only is the scheduling problem given such requirements hard, inferring them in the first place is also non trivial. Performing systemic offline experiments with various configurations and scheduling decisions, can help in learning the (possibly non - linear) mapping between application performance and low level metrics.

3.1.5 Preemption Granularity

The last challenge we would like to discuss is that the resource management decisions (especially temporal CPU allocation) must take into account the fact that certain components should be preempted only at specific times in their execution. For example, if two components A, B use shared memory or any common resource using locking mechanisms, it might not make sense to preempt A while it has acquired the lock, since the other component B will not be able to run even if we were to allot cpu time to B (because it won't be able to access the shared memory), and this can lead to wastage of cpu cycles. We observed this in the VR application, where in both TW and Render components use the OpenGL server [69], which internally uses locks to coordinate among various tasks.

Note that it might be interesting to explore managing multiple resources simultaneously for such applications, but for this work we assume that the CPU is the only contended resource (i.e. all other resources are 'ample' or sufficient).

3.2 OPPORTUNITIES

The on-board compute platform for a sense – react system typically runs only a single application. This gives the resource manager full control and visibility over the system (CPU) resources and application nodes, which enables making system-wide decisions.

Moreover, unlike many other systems, we can actively control the input load in the system by controlling the rate at which the sensor nodes sense the environment, or the rate at which other components produce new outputs. This adds another dimension to the scheduling decisions, and differentiates the problem from systems such as stream processing where the main goal is to load balance the incoming requests' load.

Finally, the long-running nature of the applications allows for periodic monitoring of resource usage, which can be useful in adapting the scheduling decisions dynamically.

CHAPTER 4: RELATED WORK

In this chapter, we shall discuss the limitations of past work on system resource management in meeting the challenges and exploiting the opportunities in sense - react systems. We look at three broad (and most closely related) categories of related work, namely real time scheduling, stream processing and offline profiling tools.

4.1 REAL TIME SCHEDULING

Before going into the literature, we first briefly describe the scheduling policies in Linux which are also modelled by many real time works.

Linux provides two policy frameworks to handle temporal CPU allocation (one of the scheduling decisions 3.1.1) - deadline based scheduling (linux SCHED_DEADLINE) or priority based scheduling (linux SCHED_FIFO). For the deadline policy, there is a deadline parameter for each component, which denotes, how quickly should the node finish execution every time it is triggered and the policy runs the component with the earliest deadline first (EDF). For the priority policy, each node has a priority parameter, and whenever a component is triggered, it will preempt all lower priority nodes. One can vary the deadline (or priority) parameters to achieve the desired temporal CPU allocation.

The real time scheduling research can be divided into two broad categories.

The first set of research works assume low level requirements are given as input, and focus on analysing whether a given set of scheduling decisions can satisfy the requirements. In particular, the paper [70] operates in Linux' SCHED_Deadline policy, and takes as input the execution frequencies as well as deadlines of all the nodes in the DAG. The main goal of that paper is then to analyse whether the given set of nodes' periods can satisfy the low level requirements as well as the nodes' deadlines. The paper [71] formulates analytical model for chain level metrics for robotics applications, but they do so under the assumption of a non preemptive, SCHED_DEADLINE based scheduling policy, with core allocation as an input. The paper [72] attempts to co-schedule both CPU and GPU for computer vision applications, but takes as input the execution frequency of all the components. The paper [73] models components having loops and switches, and analyses whether a given set of node periods and deadlines will satisfy given low level requirements on a single core. The paper [74] analyses low level metrics for a set of chain DAGs, with given periods, deadlines and degrees of parallelism.

In summary, since these works only analyse feasibility for a given set of scheduling deci-

sions. It is unclear how to extend these approaches to actually find the optimal decisions. A trivial approach of using search algorithms to explore the space of all possible scheduling decisions might not scale well with the size of the DAG. (and consequently the exponential number of scheduling decisions)

The second set of research works try to optimize a subset of the scheduling decisions. Davare et al [18] operates in the static priority based scheduling (i.e. Linux SCHED_FIFO), and tries to find the best set of frequencies for each node, with the decision of the node-to-core assignment and the priority ordering for all the nodes, as an input. ROSCH [21] also uses fixed priority based scheduling, and given the rates of the sensor nodes as an input, it applies a heuristic based approach to assign cores and priorities to the nodes in the DAG, but does not take into account the heterogeneous low-level requirements for different components. The papers [19, 20] take node rates as an input, and solve for node priorities and the cores to node assignment in the static priority scheduling paradigm. To summarize, all these approaches assume some part of the scheduling decisions as their input.

To summarize, none of the works listed above handles the challenge of inferring low level requirements 3.1.3, and just focus on solving the scheduling problem assuming that the requirements (e.g. constraints on latencies / throughputs) are given. The research also does not account for dynamicity of any kind (3.1.2), and also do not consider preemption granularity. Lastly, none of the papers evaluate the impact of scheduling decisions on application level performance.

4.2 STREAM PROCESSING

Stream processing systems resemble sense - react systems in that they both have pipelines of tasks, which need to process incoming data (from sensors in sense-react systems).

Many works in this area use back pressure - based techniques [25, 26, 27, 28, 75], which basically reduce the source's sending rate if a downstream component cannot keep up with it. SEDA [24] and Streamscope [76] model a single chain and DAG of components respectively, with each component maintaining a queue of incoming inputs. The approaches in both the papers control the number of threads a component is allowed to parallelize to, based on various parameters such as current queue size. All these papers make local decisions (per component or per edge), which may be suboptimal because they do not consider the effects of such decisions at a global level.

There is some research on making system-wide resource management decisions decisions (e.g. [23, 77, 78]), but they focus on resource allocation or task scheduling to handle a given input 'load' or data, the scale of which may vary. For example, handling client's requests

at a server. But, the distinctive feature of sense - react pipelines is that we can control the incoming load, in the sense that the rate at which we want to sense the environment is a scheduling decision that we can make.

A few other works look at input load shedding to avoid system overload (in scenarios where a component cannot keep up due to increased compute time or the input load is too high), but do not simultaneously optimize other scheduling decisions such as resource allocation and task ordering (e.g. [22, 79]).

More generally, such mechanisms are designed for a different application domain and environment (e.g. query processing in cloud clusters), and (a). do not co-optimize all the scheduling decisions, and (b). do not optimize for the application-level goals of sense - react systems.

4.3 OFFLINE PROFILING

There has been research in using extensive offline experiments to auto tune the configuration parameters for problems such as video analytics [32], stream processing systems [30, 31], cluster sizing in analytics workloads [33]. OpenTuner [29] is a general framework for program autotuning. These works use search techniques, such as gradient descent to intelligently sample the space of all parameter values so as to limit the amount of profiling experiments required, and choose the configuration that gives the best performance. While such techniques bypass challege 3.1.3, they do not account for dynamicity of various kinds - one might need to repeat the (expensive) profiling experiments if there are changes in the environment or resource availability / consumption.

CHAPTER 5: RESOURCE MANAGER DESIGN

5.1 PROBLEM FORMULATION

We model our scheduling problem as meeting constraints on and/or optimizing a weighted linear combination of the low-level metrics along different chains in a DAG. In particular, we support the following metrics - throughput of each component and throughput, latency and RT along a chain (as defined in 3.1.3).

Our approach, Jenga, therefore, requires an input objective function in the form of constraints or weights across these metrics for an application. An example objective for the navigation DAG could be to minimize the weighted sum of response time along different chains, with a higher weight for the local chain to prioritize collision avoidance. As discussed in 3.1.3, such requirements will need to be specified by domain experts / application developers, and we discuss the specific requirements we use for different applications in evaluation (§7). But, we do believe that specifying preferences and constraints on low level is more intuitive and generalizable than directly setting system configuration parameters that could vary with resource usage and availability. In future, we plan to explore whether the mapping from application-level performance goals and the lower-level per-chain metrics can be learnt via offline profiling.

Given an input objective function for an application DAG, Jenga makes CPU scheduling decisions so as to optimize the specified objective and meet the constraints. We allow the constraints to be soft, i.e. if the scheduler cannot meet them, it prints a warning, and aims to meet a looser (scaled up) set of constraints.

We divide the application DAG into *subchains* and make the scheduling decisions at the granularity of each subchain. Each subchain is a series of nodes that run at the same rate in an event-driven manner (with the output from one node triggering the next). Two components A,B can be combined into a single subchain if (a) node B is the only node that consumes data incoming from node A, and there's no value in re-running node B unless there's a new output from A, regardless of B's other inputs, or (b) a particular chain's response time (or latency) constraint is very tight, in which case making a subchain helps by allowing lower delays due to asynchrony between the nodes. Lastly, a node which has drastically unique requirements, such as map callback which 'streams' laser scan inputs, we consider it as an independent subchain.

For each subchain, Jenga must determine: (i) the number of CPU cores that must be assigned to it, (ii) the degree of per-node parallelism (for multi-threaded nodes), (iii) the

rate at which the subchain (source node) is triggered, and (iv) the temporal allocation of CPU across subchains sharing the same core. This is a hard scheduling problem. Even a DAG scheduling involving a small subset of these dimensions has been proven to be NP-hard [62]. We tackle it by adopting an hierarchical approach, where we break our scheduling decisions into two stages – the first stage determines the mapping between subchains and cores (§5.2), and the second stage makes scheduling decisions at per-core and per-subchain granularity (§5.3). We begin with outline our approach under the assumption that the computation time at each node and the number of cores is constant, and discuss how we handle dynamicity in §5.4.

5.2 STAGE I: CORE ALLOCATION

For a DAG with n subchains on a system with k cores, our first stage of optimization outputs a boolean matrix of size $n \times k$, where an element a_{ij} is 1 if subchain i is allowed to execute on core j. We use c_x to denote the compute time of node x, and $x \in SC_i$ is the set of all nodes which belong to subchain i. We assume that all cores are homogeneous, i.e. equally powerful, but our approach can easily be extended to handle heterogeneous cores.

Assumption A1. We make a key assumption for tractability: each subchain either runs alone on a certain number of cores or shares a single core with other subchains (i.e. two or more subchains do not share two or more cores). This helps us in both solving the problem and simplifying the implementation of the schedule, since each CPU core has an independent run queue to store its tasks, in Linux.

We further make two approximations in this first stage:

Approximation Ap1. When multiple subchains share a core, each subchain gets an equal share. This was inspired by the design of Linux' Completely Fair Scheduler, which ensures fairness by distributing the CPU time equally among tasks.

Approximation Ap2. When a subchain is assigned multiple cores, all of its nodes have the same degree of parallelism (denoted by q_i for subchain i). If the subchain allow parallelism (denoted by P_i) [1], we assume that the execution time for all the nodes in the subchain scales perfectly with q_i (i.e. $c'_i = c_i/q_i$).

We forgo these approximations when making finer-grained scheduling decisions in §5.3. We develop analytical models to compute per-chain latency, throughput, and response times as a function of a_{ij}. Combining with the approximations, we are able to formulate a Mixed Integer Linear Program (MILP) that solves for a_{ij} so as to minimize the specified objective

[1] true if all nodes in the subchain are parallelizable

function and meet constraints. We now discuss the formulation in detail. Let C denote the number of chains in the DAG. We denote by l_c and t_c the latency and period (i.e. reciprocal of throughput) along a chain, and by p_s the period of subchain s, and by $|c|$ the length of chain c. M denotes a very large number (we use 50,000 in our implementation).

$$\min \sum_{c=1}^{C}(w_{1c} * l_c + w_{2c} * t_c) + \sum_{s=1}^{n} w_{3s} * p_s \tag{5.1}$$

$$\text{s.t.} \sum_{j=1}^{k} a_{ij} \geq 1.0 \qquad \forall i \tag{5.2}$$

$$\sum_{i=1}^{n} a_{ij} \geq 1.0 \qquad \forall j \tag{5.3}$$

$$\sum_{j=1}^{k} a_{ij} \geq 1.0 - M * (1 - x_i) \qquad \forall i \tag{5.4}$$

$$\sum_{j=1}^{k} a_{ij} \leq 1.0 + M * x_i \qquad \forall i \tag{5.5}$$

$$y_{ij} \leq a_{ij} \qquad \forall i, j \tag{5.6}$$

$$y_{ij} \leq x_i \qquad \forall i, j \tag{5.7}$$

$$y_{ij} \geq x_i + a_{ij} - 1 \qquad \forall i, j \tag{5.8}$$

$$\sum_{i=1}^{n} a_{ij} \leq 1 + M(1 - y_{ij}) \qquad \forall i, j \tag{5.9}$$

$$\sum_{i=1}^{n} a_{ij} \geq 1 - M(1 - y_{ij}) \qquad \forall i, j \tag{5.10}$$

$$z_{ij} \leq M(a_{ij}) \qquad \forall i, j \tag{5.11}$$

$$z_{ij} \leq p_i \qquad \forall i, j \tag{5.12}$$

$$p_i - z_{ij} \leq M(1 - a_{ij}) \qquad \forall i, j \tag{5.13}$$

$$\sum_{j=1}^{k} z_{ij} \geq (max_{x \in SC_i} c_x) * b_i \qquad \forall i \tag{5.14}$$

$$1 - Mx_i \leq b_i \leq 1 + Mx_i \qquad \forall i \tag{5.15}$$

$$\sum_{j=1}^{k} z_{ij} \geq \sum_{x \in SC_i} c_x \qquad \forall i \tag{5.16}$$

$$w_{ijl} \leq a_{ij} \qquad \forall i, j, l \tag{5.17}$$

$$w_{ijl} \leq a_{lj} \qquad \forall i,j,l \qquad (5.18)$$

$$w_{ijl} \geq a_{ij} + a_{lj} - 1.0 \qquad \forall i,j,l \qquad (5.19)$$

$$p_i \geq (\sum_{x \in SC_i} c_x) * (\sum_{l=1}^{n} \sum_{j=1}^{k} w_{ijl}) - M x_i \qquad \forall i \qquad (5.20)$$

$$\text{if not } P_i : b_i = \sum_{j=1}^{k} a_{ij} \qquad \forall i \qquad (5.21)$$

$$ex_i \geq p_i - M x_i \qquad \forall i \qquad (5.22)$$

$$ex_i \geq \sum_{x \in SC_i} c_x \qquad \forall i \qquad (5.23)$$

Equation 5.1 represents the objective function, which is a weighted linear sum of the low level metrics. The parameters $w_{1c}\&w_{2c}$ represent the weights for the latency and period of chain c respectively. w_{3s} represents the weight for the period of subchain s.

Eqn. 5.2 and 5.3 represent that each subchain (core) should be assigned at least one core (subchain) respectively.

x_i denotes whether a subchain gets more than one cores, and Eqn 5.4 and 5.5 represent this. Equations 5.6 - 5.8 enforce that $y_{ij} = a_{ij}\&\&x_i$, since y_{ij} represents whether subchain i is on multiple cores (x_i) and core j is one of them (a_{ij}). Equations 5.9 - 5.10 denote that if $y_{ij} = 1$, then there should be only one subchain on core j. All these eqns together enforce our assumption A1.

We analytically lower bound the period of each subchain based on the a_{ij} parameters. Lets take two cases:

Case 1. If a subchain gets $\sum_{j=1}^{k} a_{ij}$ cores to execute, then $q_i = \frac{\sum_{j=1}^{k} a_{ij}}{b_i}$ would be the degree of parallelism i.e. we allow each node in the subchain to use at most q cores. As per approximation Ap2, we assume each node's compute time scales perfectly with q [which we forgo in §5.3], and and the period is given by $p_i = max(max_{x \in SC_i}(c_x/q_i), \Sigma_{x \in SC_i}(c_x)/b_i)$. This formula was inspired by our theoretical result of optimal response time for a single chain on multiple cores, explained in more detail in §5.3. We encode this formula (eqn 5.14, 5.16) in our formulation by making extra variables $z_{ij} = p_i a_{ij}$ (Eqns 5.11 - 5.13). Eqn 5.15 signifies that $q_i = b_i = 1$ if the subchain is on a single core, eqn 21 states that $q_i = 1$ and consequently b_i is the number of cores alloted to the subchain, if it is not parallelizable. The MILP solver will make the trade off of q_i while solving for a_{ij}; a larger q_i may reduce per-node processing times, but also reduces the degree of pipelining along the subchain. Note

that we didn't directly use the q_i parameter in our constraints so as to be able to linearize the formulation.

Case 2. If a core has $s = \sum_{i=1}^{n} a_{ij}$ subchains assigned, then, due to approximation Ap1, each subchain i gets $1/s$ share of the CPU, and hence will finish one execution in time $(\sum_{x \in SC_i} c_x) * s$. This is represented in eqn 5.20, where in, if subchain i doesn't have > 1 cores, we lower bound the chain's period by the product of the compute time of the subchain multiplied to the total number of subchains on its core. To help express this constraint in a linear form, we make extra variables of the form w_{ijl}, which represent if subchain i and l share core j [represented in Eqns 5.17 - 5.19]. Note that we forgo this approximation in §5.3, wherein we design a fine grained scheduling policy for each such core.

The variable ex_i represents the execution time of a subchain i, i.e. how long does it take to finish one full execution & produce an output. It is at least the total compute time of the subchain (eqn 5.23), and is equal to the subchain's period if it is on a single core (eqn. 5.24) [one could get a tighter bound but we use this approximation].

Let chain c be denoted by $s_{c0}, s_{c1}...$ Eqn 5.24 and 5.27 describe the chain throughput and latency as a function of other variables. We add constraints corresponding to the following : LT_x (LT_c) and UT_x (UT_c) represent the lower and upper bounds on the throughput of node x (chain c) respectively. LL_c and UL_c similarly represent the lower and upper bound constraints on the chain c's latency. We can also add constraints/ weights on the chain's response time, which we approximate as $l_c + t_c$. Eqns 5.25, 5.26 and 5.28 represent these constraints.

$$t_c \geq p_i \qquad \forall SC_i \in C_c \tag{5.24}$$

$$LT_c \leq t_c \leq UT_c \qquad \forall c \tag{5.25}$$

$$LT_x \leq p_i \leq UT_x \qquad \forall x \in SC_i \tag{5.26}$$

$$l_c \geq ex_{c0} + \sum_{r=1}^{|c|} ex_{cr} + p_{cr} \qquad \forall c \tag{5.27}$$

$$LL_c \leq l_c \leq UL_c \qquad \forall c \tag{5.28}$$

5.3 STAGE II: PER CORE & PER SUBCHAIN SCHEDULING

Given the subchain to core mappings from our first optimization stage, the next stage makes finer-grained scheduling decisions for each subchain and core. There are two distinct cases to consider:

Single subchain on one or more cores. We shall now extend the approach mentioned in

§5.2 (Case 1). Considering a subchain of nodes $\{n_1, n_2, \ldots n_m\}$ with k assigned cores. The scheduler must determine the rate (or the time period p) of the source node (n_1) and the degree of parallelism for each node, such that the response time is minimized. We allow each node in the subchain to use at most q cores (only a subset may be designed to use all q), and set $p = max(max_{j=1}^{m}(c_j^q), \Sigma_{j=1}^{m}(c_j^q)/\lfloor k/q \rfloor)$, where c_j^q is n_j's computation time with at most q cores [2]. We have proved that this rate allocation achieves response time (for this particular subchain) within $2\times$ the optimal, and equal to the optimal for $q = 1$ [62]. The value of q ranges from 1 to k; a larger q may reduce per-node processing times, but also reduces the degree of pipelining along the subchain. The scheduler iterates over all k choices for this subchain, and selects the value of q that results in lowest response time for this subchain.

Multiple subchains on a single core. For subchains $s_1, s_2 ... s_n$ assigned the same core Q_j, the scheduler must determine the temporal allocation of CPU time across them. We define a variable f_i for each subchain, which denotes two things: a) the subchain gets $\Sigma_{x \in SC_i}(c_x) * f_i$ CPU time per period, b) the subchain finishes one execution per $1/f_i$ periods. The total period of this schedule then comes out to be $\sum_{i \in C_j} f_i * (\Sigma_{x \in SC_i}(c_x))$. The execution of each (fractional) subchain within a period follows topological ordering. Such a schedule ensures that the CPU core is fully-utilized, while the fraction variables tune the core allocation across subchains. We distinguish between subchains (or nodes) which have constraints on average throughput (e.g. streaming nature such as GML) from others, by allowing f to take arbitrary values for such nodes, but constraining it to be integral - reciprocal (i.e. $1/f_i$ is an integer) for other nodes. This is because, $1/f_i$ being integral allows the scheduler to control the exact throughput of the subchain.

We approximate the per chain metrics as an analytical function of these fraction variables, and formulate a Geometric Programming problem to compute these variables such that the specified objective function is optimized. We now describe our formulation. Note that we make a single GP problem to solve for temporal CPU allocation of all the cores. Let SN denote the set of subchains of the streaming nature, and let Q_j denote core j, and SQ_j denote the set of all the subchains sharing core j.

$$\min \sum_{c=1}^{C} (w_{1c} * l_c + w_{2c} * t_c) + \sum_{s=1}^{n} w_{3s} * p_s \tag{5.29}$$

$$\text{s.t.} f_i \leq 1.0 \qquad \forall x \notin \text{SN} \tag{5.30}$$

$$p_j^q \geq \sum_{i \in SQ_j} (f_i * \Sigma_{x \in SC_i}(c_x)) \qquad \forall j \tag{5.31}$$

[2] Note that we are not assuming perfect scaling here, and can use the actual/empirical compute time of each node given q cores

$$p_i \geq \frac{1}{f_i} * p_j^q \qquad \forall i \in SQ_j \tag{5.32}$$

$$t_c \geq p_i \qquad \forall SC_i \in C_c \tag{5.33}$$

$$LT_x \leq p_i \leq UT_x \qquad \forall x \in SC_i \tag{5.34}$$

$$l_c \geq p_{c0} + \sum_{r=1}^{|c|} 2 * p_{cr} \qquad \forall c \tag{5.35}$$

$$LT_c \leq t_c \leq UT_c \qquad \forall c \tag{5.36}$$

$$LL_c \leq l_c \leq UL_c \qquad \forall c \tag{5.37}$$

p_j^q denotes the period of core j, and its formula is given by Eqn 5.31. Eqn 5.32 represents that each subchain finishes one full execution in $1/f_i$ periods of its core.

Eqns 5.33 and 5.35 describe the formulae for chains' throughput and latency (similar to §5.2), and Eqns 5.36, 5.37 represent lower and upper bound constraints on these chain level metrics. We similarly can add constraints on node throughput using eqn 5.34. Note that the lower bound constraints are not tight here, i.e. $p_i \geq LT_x$ does not necessarily imply that the the subchain's period (eqn 5.32) will also be more than LT_x. This is mostly because geometric programming only supports less than constraints to maintain convexity.

5.4 HANDLING DYNAMICITY

Coarse Grained Dynamicity. We handle dynamicity by recording the computation time across all nodes and tracking the number of available cores over time. We re-compute the stage II scheduling decisions (§5.3) periodically (every 5s in our implementation) using the 95%ile computation time for each node measured over the previous 50 values in our implementation (We support multimodal compute times by taking the weighted sum of 95%ile compute times across the different modes). We re-invoke the stage I optimization (§5.2) less frequently (once every 20s in our implementation). We chose these periods so as to keep the overhead of re-solving low.

Fine Grained Dynamicity. We are exploring using priorities or precedence as a way to handle fine grained variations, e.g. sudden spikes in compute times. For instance, we can specify if a node B wants to wait for a new output from another node A [3]. This will allow node B to take up some of the cpu time assigned to node A if needed. We can add extra constraints in our optimization problems to reflect this as well.

[3]This is almost a middleground between B being independently frequency driven and being synchronous w.r.t. A

CHAPTER 6: IMPLEMENTATION

We implement the Jenga controller as a ROS node for robotics and as an ILLIXR plugin for AR/VR, both of which use the same backend which handles all the scheduling decisions. The controller assumes that each node runs in a thread of its own.

Initialization. At the start of the application, the controller requires that all the nodes should send their thread id. For robotics, we modify ROS' ros_comm library to expose the ids of all the threads it uses under-the-hood to handle communication between nodes. The scheduler uses the thread ids for each node to control when does it get to execute, at runtime. The controller runs the stage I optimization to map nodes (and their corresponding threads) to cores. Then, for each core j with multiple subchains, the controller spawns a thread (let's denote it by ThC_j) which is assigned to handle temporal cpu allocation for core j, and, for each subchain i assigned to multiple cores, it similarly spawns a thread (let's denote it by ThS_i). Finally, it spawns an extra thread (denoted by $ThOpt$) to handle updating the optimal scheduling decisions dynamically.

Runtime. At runtime, each scheduler thread ThS_i triggers the first (source) node of the subchain i at the analytically computed time period, and informs the nodes of the subchain of the computed degree of paralellism. Each scheduler thread ThC_j enforces the fractional schedule, by (a). sending a trigger to the first node in each (non-streaming) subchain s_y every $1/f_y$ periods, and (b). in each period, iterating over all the subchains, and for subchain s_y, assigning Linux' SCHED_FIFO policy with high priority to all the thread ids of (all the nodes in) the subchain and low priority to all threads of other nodes, for exactly $(\sum_{x_i} c_x) * fi$ or $(\sum_{x_i} c_x) * Fi$ time. Figure 6.1 demonstrates this schedule for the navigation application. We make one modification to this - for subchains with optimal fraction = 1, we wait for their execution to finish before moving on with the schedule. Each subchain is internally purely event driven [1]. Lastly, each node sends its compute time to the scheduler periodically.

MILP & GP Implementation. The thread $ThOpt$ periodically re-solves for the new optimal scheduling decisions, based on the latest compute time estimates of all nodes (it uses the 95%ile compute time of the last 50 values). It re-computes Stage I every 20s, and Stage II every 5s. Both the optimization problems (MILP for stage I and GP for stage II) have been implemented using the Mosek Fusion library in C++. We added extra constraints to the stage II optimization, namely (i). scale the period of each core by 1.05 to adhere to Linux' constraint of allowing real time processes to only use 95% CPU [80], and (ii). use fractions for subchains s.t. the amount of time alloted to the subchain per period

[1] i.e. each node waits on its predecessor to finish

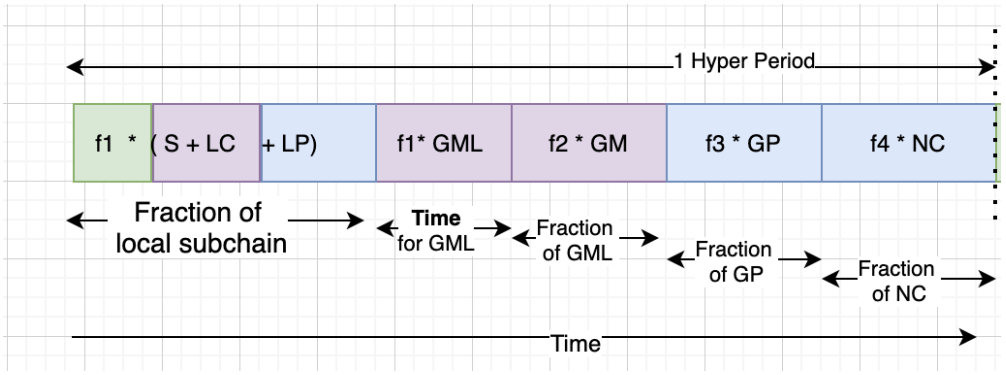

Figure 6.1: DAG representing the face tracking robot application

is atleast 1ms, i.e. $(\sum_{x_i} c_x) * fi \geq 0.001$, this was to reduce the overhead of preemption and priority changing per subchain, per period (based on the mechanism described in the Runtime section above). For the navigation DAG with two (three) cores, the solvers for stage I and stage II take 50ms (150ms) and 16ms (15ms) respectively on an AWS EC2 instance of type m4.4xlarge [2]. Note that we periodically re-solve at a rate so as to use less than 5% CPU for these computations.

Note that we had tried using SCHED_DEADLINE directly to enforce the fractional schedule, but it turned out to be quite inflexible in allowing fine grained control of the schedule (e.g. in terms of waiting for a subchain to finish, scheduling at subchain granularity), though it might've been more efficient.

[2]The time taken for stage - I scales with the number of cores since the number of variables a_{ij} also scales. But, for stage - II, the number of variables only depends on the DAG and not cores

CHAPTER 7: EVALUATION

We will now discuss the evaluation setup for the three applications, as well as their specific requirements, constraints and the corresponding results.

7.1 FACE TRACKING ROBOT

We start with evaluating the face tracking application, which has a simple DAG comprising of a single chain.

7.1.1 Setup

We use Gazebo [81] to simulate a dummy moving along a square path centered at a robotic camera that can rotate at a velocity of upto 3.2rad/s ([82, 83]). Gazebo feeds camera inputs into detection and planning nodes implemented in ROS, which in turn feed the velocity output back to the simulated camera. We make a few modifications to this basic setup to better model realistic timings.

We run the whole setup (i.e. the simulator as well as the application) on an AWS EC2 instance (m5.4xlarge). Gazebo runs on 5 reserved cores, and sends camera inputs as fast as it can to a shim source node, which then publishes the latest input at the specified frequency. We add a node that models the time taken for post-processing or formatting camera inputs (measured as 25ms using usb-cam module [84]). Since the algorithm to detect the dummy is trivial (detects a green rectangle since the dummy has a green shirt), we model realistic timings by augmenting our detection node – every time it processes a simulated frame, it also runs a single-cascade HAAR algorithm [37, 85] on a frame drawn from a real video data-set [86, 87] (the time taken for this varies between 55-60ms). The planning node (which takes 1ms) tracks the dummy using the detection output from the simulated frame. We experiment with varying source frequency and speed at which the dummy moves, as well as with different number of cores k made available to the application (that excludes Gazebo).

7.1.2 Requirements

Since there is only a single chain, we model it as a single subchain, and hence the scheduling problem boils down to the case of a single subchain on one or multiple cores (from §5.3). We model the problem as having an objective of minimizing the response time of the chain.

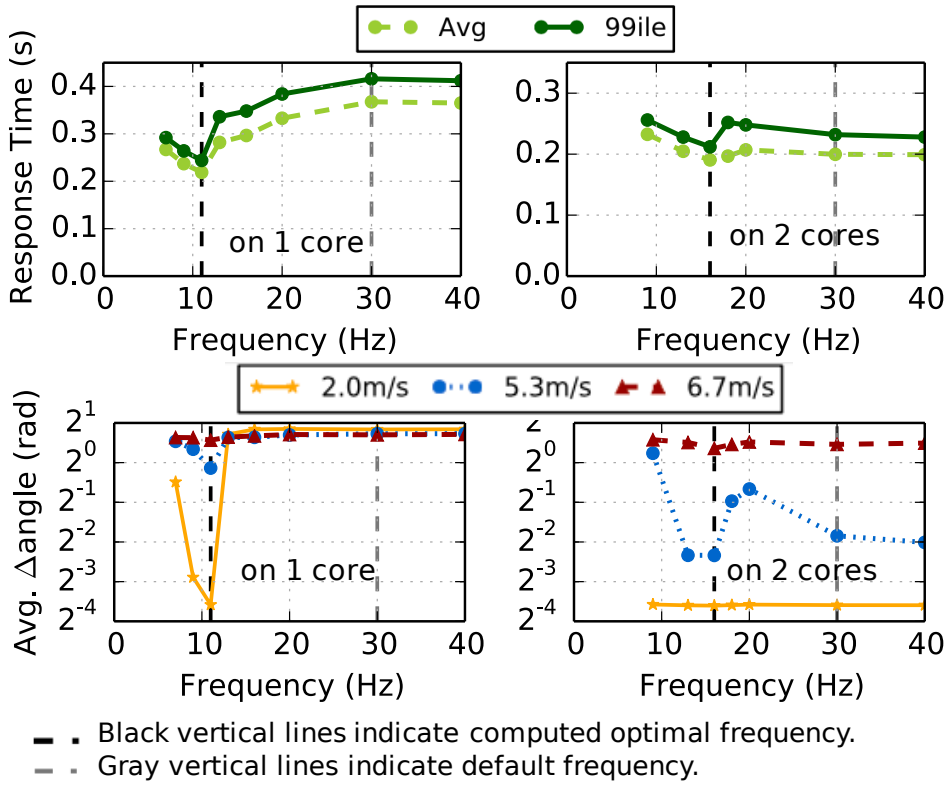

Figure 7.1: Response time (top) and tracking performance for different object speeds (bottom) as source frequency is varied on one and two core systems.

7.1.3 Results

Metrics. We measure the tracking performance of the robot as average angular distance (δangle) between the camera's orientation and the position of the dummy (a lower value implies better performance).

Figure 7.1 shows the response time and tracking performance on a system with one and two cores, for a range of source frequencies (with the computed optimal and default indicated by vertical lines). The optimal here is based on our theoretical analysis for a single chain DAG which minimizes the chains' response time. The compute capacity is the bottleneck with one core, while the slowest node is the bottleneck with two cores. The top graphs in Figures 7.1 reveal that is the lowest at the analytically optimal source frequency (computed using the tail node processing times).

In terms of the performance metric, we draw the following key observations: (i) Tracking performance strongly correlates with the response time, confirming the impact of low-level metrics on application performance. (ii) The computed optimal frequency performs the best, outperforming the default value. This shows the impact of scheduling on application performance. (iii) In general, it is harder to track a faster moving object. The optimal

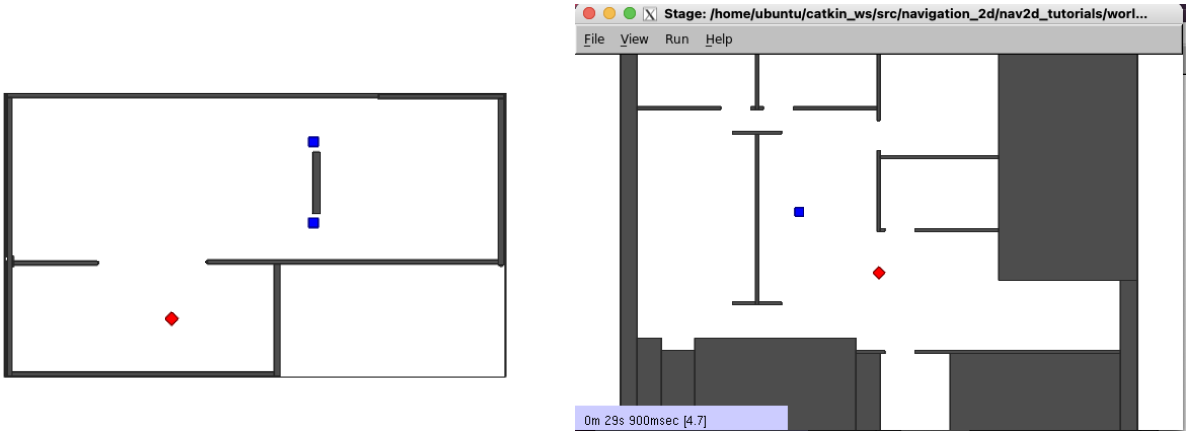

Figure 7.2: (Left) Map-I : A small map with 2 dynamic obstacles (blue), which start moving towards the opposite wall (to block the robot's path) when the robot (red) arrives near. (Right) Map-II : A map with multiple rooms and one static obstacle.

frequency is able to track the object for a wider range of speeds.

7.2 ROBOT NAVIGATION

7.2.1 Setup

We now use the navigation application described in §**??** for evaluation. We use Stage [88] to simulate a P3AT robot [89]. The simulator feeds laser scans and odometry into the application, which is implemented on top of ROS, and the application feeds the robot's velocity back into Stage. Fig 7.2 shows the two maps we use for the robot to explore. The robot starts at a fixed location in the map, and needs to explore the whole area, while avoiding collisions with the obstacles.

7.2.2 Requirements

We obtained constraints on response times for all the chains using experiments with the default codebase on a variety of frequencies. We model the objective function as weighted sum of these 4 response times, with weights proportional to the constraints. We make S - LC - LP a single chain considering its very tight constraint to avoid obstacles. We also add the following application specific requirements into our optimization formulation :

- Since GM does not execute unless there is a new output from GML, its period is constrained by GML's period. Also, due to GML's logic of discarding laser scans with

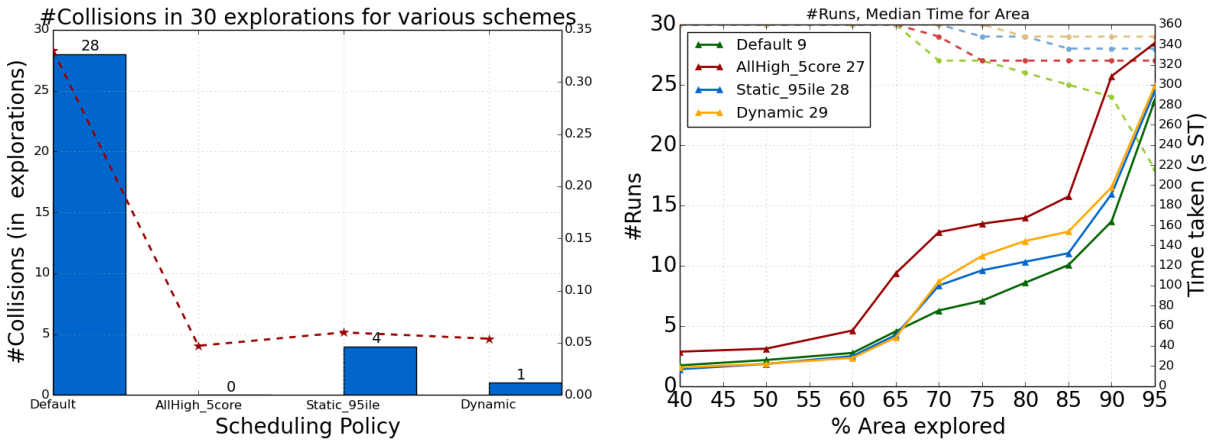

Figure 7.3: Results for 30 explorations on Map - I (Left) Bars denote Collisions and the dotted line denotes the median RT for local chain for different policies. (Right) For every % area covered, the dotted lines plot how many explorations covered that much area, and the solid lines plot the median (simulated) time taken to cover that area. The number in the legend represents the number of explorations which had 0 collisions and covered 90% area.

no new information, it only produces new outputs at 1Hz on average and 1.5Hz in 25%ile. So, we add a constraint on GM to be slower than 1.5Hz, and GML's period used in the chain's latency formula is also constrained to be slower than 1.5Hz.

- Stage publishes laser scans and odometry at 50Hz (we have sped up simulation by 5x). We add a constraint on GML's period being atleast 20ms, and its weight in the objective function is non zero.

- The application had logic to not run GP faster than 1Hz, which was to avoid thrashing, as described in §3.1.3. We hence add this as a constraint in our formulation.

7.2.3 Results

Metrics. We consider two key application metrics – the number of collisions, the rate of exploration, and the proportion of explorations which successfully explore the whole map. We measure the collisions as reported by the simulator, and the rate of exploration based on the GM' output map's size.

We evaluate two scenarios – a robot tasked with exploring (i) 380sq.m. total area with dynamic obstacles, and (ii) 550sq.m. total area with one static obstacle (Fig 7.2). We run both the experiments on an AWS EC2 machine (m4.4xlarge), with stage allocated 6 cores, and the application allocated one core (the scheduler included). We compare against two

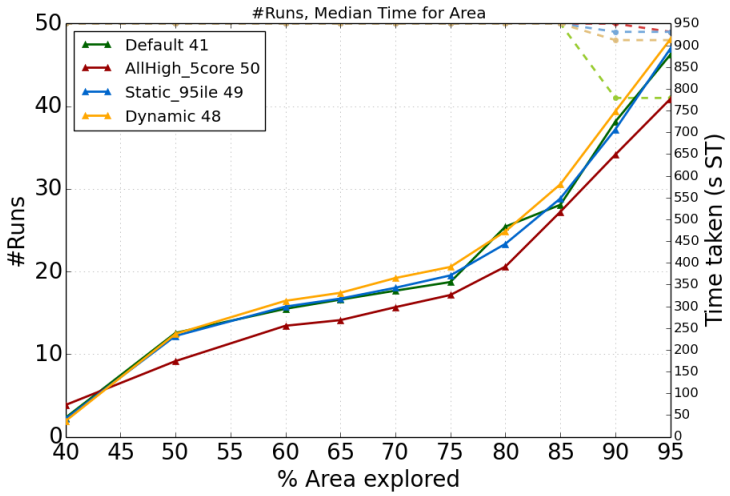

Figure 7.4: Results for 50 explorations on Map - II for different schemes. For every % area covered, the dotted lines plot how many explorations covered that much area, and the solid lines plot the median (simulated) time taken to cover that area. The number in the legend represents the number of explorations which had 0 collisions and covered 90% area.

Figure 7.5: The room and the headset view for the VR application

baselines: (i) hand-tuned node frequencies set as default by application developers, relying on default OS policies for scheduling, and (ii) an all high setting, where each subchain is on a separate core (total 5 cores), running at 100Hz. We compare these policies with two variations of Jenga- (a) Static_95ile, where in the stage II optimizations are solved at initialization time assuming the knowledge of execution time of all nodes, (b) Dynamic, where the controller periodically re-computes the optimal solution based on observed compute times.

Map - I. Figure 7.3 shows the results for the small map with dynamic obstacles. We draw the following key observations: (i) Jenga Static and Dynamic, as well as AllHigh avoid collisions more effectively than the default configuration in the obstacles scenario (also reflected by Runs which cover $> 90\%$ area with no collisions which is better by 3x). This stems from the fact that these three schemes much better response time for the local chain than the default

Approach	Scheduling Inputs	Time
Jenga	Compute time of each node	30-60ms (C++ Mosek)
Davare et al [18]	Compute time of each node, Node priorities core allocation.	1.2s (Python)
Verucchi et al [70]	Compute time of each node, Node Rates and deadlines.	6min-1hr (C++)

Table 7.1: Comparing the different approaches on what inputs they require (excluding low level requirements which are an input for all the above), and how long they take, in terms of solving the resource management problem for the robot navigation application on a single core.

scheme. (ii) The number of explorations which cover atleast 90% area is also the worst for Default. (iii) The rate of exploration is slightly worse for our policies but even worse for AllHigh. This highlights that the 'obvious' optimal strategy of running every node at the highest possible frequency on separate cores might not perform well.

Map - II. Figure 7.4 shows the results for map - II with just one static obstacle. There were no collisions in any of the schemes hence we only plot the area metric. We make the following observations - (i) Our exploration rate is similar to default (We were not expecting this and are investigating reasons for no performance improvement), but AllHigh does do slightly better. (ii) The number of explorations out of 50 runs, that cover more than 90% area is better for Jenga Static and Dynamic, and AllHigh, by 19%, 17% and 22% respectively.

We had tried to evaluate for the related works as well - specifically, Davare et al [18] and Verucchi et al [70]. Table 7.1 shows that these approaches (i) only solve for a subset of the scheduling decisions, (ii) take a long time to execute.

7.3 VIRTUAL REALITY

7.3.1 Setup

We run a VR application for a user looking around in a room (Fig 7.5). We use a publicly available IMU and camera dataset to simulate the sensors. We run the VR application DAG on a single core of 3.7GHz frequency.

7.3.2 Requirements

We model the IMU - Int - TW as one subchain, but allow IMU and Int to run at a fixed frequency of 200Hz (since that is the frequency in the sensor dataset).

As discussed in §3.1.3, the frequency of Timewarp needs to be exactly vsync, which for this particular hardware setup is 120Hz (8.33ms). We use constraints on the IMU - Int - TW chain based on domain expertise.

7.3.3 Results

We compared the performance of the Default system (hand tuned parameters for the specific hardware) and Jenga system based on subjective visual metrics, and both the systems seemed to have a similar level of smoothness in the rendered video. But, Jenga was able to automatically meet the vsync requirement (mean and median error (i.e. difference between vsync and when the scheduler executes TW) is 0.03ms), which had been achieved in the default scenario by forcing TW to sleep for a manually tuned time [hardware - specific].

We are currently working on evaluating these applications for multi core scenarios, and possibly extending to more complicated applications.

CHAPTER 8: CONCLUSION & FUTURE WORK

While our work highlights the challenges involved in CPU scheduling for sense - react applications, it also shows the promise of dynamic resource management in improving a robot's performance, opening multiple interesting directions for future research.

Inferring Low – level Metrics. Jenga requires domain expertise to specify an objective function in terms of low-level requirements and preferences on throughput and latency. While more intuitive than tuning system configuration, it may still be non-trivial to reason about how these metrics impact application performance (as also highlighted in §3.1.3). Using a systemic and automated profiling based approach to infer the mapping between application performance and lower-level metrics, and evaluating how well such a mapping generalizes across scenarios is an interesting (and important) future direction. Moreover, the objective function itself could vary over time (e.g. being closer to certain locations may require higher throughput along certain chains). Future extensions of Jenga could support such adaptations.

General Deployment Models. Our current work considers sense - react systems with a single on-board computer, with CPU as the only contended resource. Future work can look into simultaneously managing different shared resources (CPU, GPUs, network bandwidth) in more complex multi-computer or cloud robotics systems.

REFERENCES

[1] "Roomba," https://www.irobot.com/roomba.

[2] "KiwiBot," https://www.kiwicampus.com/.

[3] "Amazon Robotics," https://www.amazonrobotics.com/#/.

[4] "Hansen Medical," https://www.aurishealth.com/hansen-medical.

[5] "da Vinci Si Surgical Robot," https://med.nyu.edu/robotic-surgery/physicians/what-robotic-surgery.

[6] "ROSA Knee System," https://www.zimmerbiomet.com/medical-professionals/knee/product/rosa-knee.html.

[7] "How Drones Are Being Used to Battle Wildfires," https://www.smithsonianmag.com/videos/category/smithsonian-channel/drones-are-now-being-used-to-battle-wildfires/.

[8] "Waymo," https://waymo.com/.

[9] "Tesla Autopilot," https://www.tesla.com/autopilot.

[10] R. Aggarwal, T. P. Grantcharov, J. R. Eriksen, D. Blirup, V. B. Kristiansen, P. Funch-Jensen, and A. Darzi, "An evidence-based virtual reality training program for novice laparoscopic surgeons," *Ann Surg*, vol. 244, no. 2, pp. 310–314, Aug 2006.

[11] S. Krohn, J. Tromp, E. M. Quinque, J. Belger, F. Klotzsche, S. Rekers, P. Chojecki, J. de Mooij, M. Akbal, C. McCall, A. Villringer, M. Gaebler, C. Finke, and A. Thöne-Otto, "Multidimensional evaluation of virtual reality paradigms in clinical neuropsychology: Application of the vr-check framework," *J Med Internet Res*, vol. 22, no. 4, p. e16724, Apr 2020. [Online]. Available: https://www.jmir.org/2020/4/e16724

[12] M. Huzaifa, R. Desai, S. Grayson, X. Jiang, Y. Jing, J. Lee, F. Lu, Y. Pang, J. Ravichandran, F. Sinclair, B. Tian, H. Yuan, J. Zhang, and S. V. Adve, "Exploring extended reality with illixr: A new playground for architecture research," 2021.

[13] M. McGuire, ""exclusive: How nvidia research is reinventing the display pipeline for the future of vr, part 1," 2017. [Online]. Available: https://www.roadtovr.com/exclusive-how-nvidia-research-is-reinventing-the-display-pipeline-for-the-future-of-vr-part-1/

[14] "LoCoBot," http://www.locobot.org/.

[15] "Terrasentia : The automated crop monitoring robot," https://blog.plantwise.org/2018/07/17/terrasentia-the-automated-crop-monitoring-robot/.

[16] "Turtlebot3 Waffle Pi," https://www.robotis.us/turtlebot-3-waffle-pi/.

[17] "TurtleBot 3 ," https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications.

[18] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 278–283.

[19] Y. Yang, A. Pinto, A. Sangiovanni-Vincentelli, and Q. Zhu, "A design flow for building automation and control systems," in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 105–116.

[20] D. Casini, P. Pazzaglia, A. Biondi, G. Buttazzo, and M. Di Natale, "Addressing analysis and partitioning issues for the waters 2019 challenge," 07 2019.

[21] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "Rosch:real-time scheduling framework for ros," in *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2018, pp. 52–58.

[22] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, p. 309–320.

[23] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The Design of the Borealis Stream Processing Engine," in *CIDR 2005*.

[24] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: Association for Computing Machinery, 2001. [Online]. Available: https://doi.org/10.1145/502034.502057 p. 230–243.

[25] "ZeroMQ," http://zeromq.org/.

[26] "RabbitMQ," https://www.rabbitmq.com/.

[27] M. Welsh and D. Culler, "Adaptive overload control for busy internet servers," in *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, 2003.

[28] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 239–250.

[29] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014, pp. 303–315.

[30] M. Bilal and M. Canini, "Towards automatic parameter tuning of stream processing systems," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3127479.3127492 p. 189–200.

[31] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2016, pp. 39–48.

[32] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.

[33] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2038916.2038934

[34] "ROS," https://www.ros.org/.

[35] "Openxr," https://www.khronos.org/openxr/.

[36] P. Goebel, *ROS By Example*, Lulu, Ed. Lulu, 2013.

[37] "ROS Face Tracking," http://wiki.ros.org/face_detection_tracking.

[38] "ROS Navigation2D Application," http://wiki.ros.org/nav2d.

[39] "TurtleBot," https://www.turtlebot.com/.

[40] "Fetch Robotics Specification," http://docs.fetchrobotics.com/FetchRobotics.pdf.

[41] G. Hu, W. P. Tay, and Y. Wen, "Cloud robotics: architecture, challenges and applications," *IEEE network*, 2012.

[42] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A survey of research on cloud robotics and automation," *IEEE Transactions on automation science and engineering*, 2015.

[43] "Rapyuta Robotics," https://www.rapyuta-robotics.com/.

[44] A. K. Tanwani, N. Mor, J. Kubiatowicz, J. E. Gonzalez, and K. Goldberg, "A fog robotics approach to deep robot learning: Application to object recognition and grasp planning in surface decluttering," *CoRR*, vol. abs/1903.09589, 2019.

[45] "Htc vive pro," https://www.vive.com/us/product/vive-pro/.

[46] " Qualcomm Snapdragon XR2 5G Platform," https://www.qualcomm.com/products/snapdragon-xr2-5g-platform.

[47] "NVIDIA Jetson: Embedded Systems for Next-Generation Autonomous Machines."

[48] "Qualcomm Snapdragon 835 Mobile Platform."

[49] "Oculus Go," https://www.oculus.com/go/.

[50] "Oculus Quest 2," https://www.oculus.com/quest-2/.

[51] "OROCOS," http://www.orocos.org/.

[52] "LCM," https://lcm-proj.github.io/.

[53] "CORBA," https://www.corba.org/.

[54] P. Fitzpatrick, E. Ceseracciu, D. Domenichelli, A. Paikan, G. Metta, and L. Natale, "A middle way for robotics middleware," pp. 42–49, 2014.

[55] "NVIDIA Isaac SDK," https://developer.nvidia.com/isaac-sdk.

[56] M. Quigley, K. Conley, B. P Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y Ng, "Ros: an open-source robot operating system," 2009.

[57] "ROS:Community Metrics Report," http://download.ros.org/downloads/metrics/metrics-report-2018-07.pdf.

[58] "Fetch Robotics," http://docs.fetchrobotics.com/FetchRobotics.pdf.

[59] "Clearpath Robotics," https://clearpathrobotics.com/.

[60] "AWS Robomaker," https://aws.amazon.com/robomaker/.

[61] "Lessons learned building a self-driving car on ROS, Cruise Automation," https://roscon.ros.org/2018/presentations/ROSCon2018_LessonsLearnedSelfDriving.pdf.

[62] "Cpu scheduling for robotics applications - theoretical results," https://drive.google.com/file/d/1mnblouvU2UMb9fdQzEQszPRWPBUHv9CH/view?usp=sharing.

[63] "Introduction to mobile robotics - slam," http://ais.informatik.uni-freiburg.de/teaching/ss12/robotics/slides/12-slam.pdf.

[64] S. Kaul, M. Gruteser, V. Rai, and J. Kenney, "Minimizing age of information in vehicular networks," in *2011 8th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*. IEEE, 2011, pp. 350–358.

[65] C. Xie, X. Zhang, A. Li, X. Fu, and S. Song, "Pim-vr: Erasing motion anomalies in highly-interactive virtual reality world with customized memory cube," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 609–622.

[66] J. Meng, S. Paul, and Y. C. Hu, "Coterie: Exploiting frame similarity to enable high-quality multiplayer vr on commodity mobile devices," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3373376.3378516 p. 923–937.

[67] "Openkarto : Travel distance parameter," https://github.com/skasperski/navigation_2d/blob/8e8366ce5440be0aca11d2d53ea1d1c33b3b9811/nav2d_karto/OpenKarto/source/OpenMapper.cpp#L2200.

[68] "Openkarto : Travel heading parameter," https://github.com/skasperski/navigation_2d/blob/8e8366ce5440be0aca11d2d53ea1d1c33b3b9811/nav2d_karto/OpenKarto/source/OpenMapper.cpp#L2201.

[69] "OpenGL," https://www.opengl.org//.

[70] M. Verucchi, M. Theile, M. Caccamo, and M. Bertogna, "Latency-aware generation of single-rate dags from multi-rate task sets," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 226–238.

[71] D. Casini, T. Blaß, I. Lütkebohle, and B. Brandenburg, "Response-time analysis of ros 2 processing chains under reservation-based scheduling," 07 2019.

[72] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, "Making openvx really "real time"," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 80–93.

[73] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 71–80.

[74] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 217–226.

[75] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14, 2014.

[76] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "Streamscope: Continuous reliable distributed processing of big data streams," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, 2016.

[77] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, "Operator scheduling in a data stream manager," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, p. 838–849.

[78] M. J. Sax, M. Castellanos, Q. Chen, and M. Hsu, "Performance optimization for distributed intra-node-parallel streaming systems," in *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, 2013, pp. 62–69.

[79] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, 2004.

[80] "Linux CPU Scheduling - Limiting the CPU usage of real-time and deadline processes," https://man7.org/linux/man-pages/man7/sched.7.html.

[81] "Gazebo," http://gazebosim.org/.

[82] "Turtlebot3 Robot Actuators," https://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/#actuators.

[83] "Turtlebot3 Robot Actuators Specifications," https://emanual.robotis.com/docs/en/dxl/x/xm430-w210/.

[84] "ROS Usb cam : Driver for USB cameras," http://wiki.ros.org/usb_cam.

[85] M. J. Paul Viola, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, 2001.

[86] G. G. Chrysos, E. Antonakos, S. Zafeiriou, and P. Snape, "Offline deformable face tracking in arbitrary videos," in *2015 IEEE International Conference on Computer Vision Workshop (ICCVW)*, 2015, pp. 954–962.

[87] J. Shen, S. Zafeiriou, G. G. Chrysos, J. Kossaifi, G. Tzimiropoulos, and M. Pantic, "The first facial landmark tracking in-the-wild challenge: Benchmark and results," in *2015 IEEE International Conference on Computer Vision Workshop (ICCVW)*, 2015, pp. 1003–1011.

[88] "Stage Simulator," http://playerstage.sourceforge.net/doc/Stage-3.2.1/.

[89] "PR2 Robot," https://robots.ieee.org/robots/pr2/.