# MOTION PLANNING WITH TASK SCHEDULING IN HETEROGENEOUS COMPUTING SYSTEMS

by

Noah B Fields

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Computer Science).

Golden, Colorado

Date _____

Signed: _____

Noah B Fields

Signed: _____

Dr. Neil T. Dantam
Thesis Advisor

Signed: _____

Dr. Mehmet E. Belviranli
Thesis Advisor

Golden, Colorado

Date _____

Signed: _____

Dr. R. Iris Bahar
Department Head and Professor, Computer Science
Department of Computer Science

# ABSTRACT

Motion planning is an important problem in many contexts of robotics. Heterogeneous computing systems in robots are able to run tasks on different processing units in varying orders, but with different impacts on the robot's state and performance. Currently existing sampling-based motion planning frameworks explore a state space through typically random sampling to create a path to a goal region, but only consider physical obstacles in their way such as walls, and do not consider the constraints of computational requirements on the path or the impacts of choosing different schedules for computation. We introduce a novel system which uses Petri nets as a modeling system on the computational requirements, and uses constraint solvers to find computation schedules for the motion planning tasks. This allows us to select motions based not only on their physical validity, but also computation-related parameters. We subdivide a space of constraints on the system into regions, enabling schedule reuse in order to improve the algorithm's efficiency. We also discuss the use of Petri nets to model another aspect of computation in a heterogeneous environment, memory contention. Our system enables us to consider physical dynamics such as heat and power in a way that prior systems are not capable. We demonstrate that our system can handle a variety of constraints of different severities, and can avoid computational obstacles more effectively than naïve planning systems which do not consider computational constraints and instead disallow regions as though they are physical obstacles.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Colorado School of Mines . . . . . CSM

Control-based Rapidly Exploring Random Tree . . . . . CRRT

Petri Net . . . . . PN

Processing Unit . . . . . PU

Rapidly-Exploring Random Tree . . . . . RRT

ACKNOWLEDGMENTS

I would like to thank and acknowledge my committee, Dr. Neil Dantam, Dr. Mehmet Belviranli, Dr. Dinesh Mehta, and Dr. Dong Chen. Thank you to my parents, Natalie and Howard Fields, and all others who have helped me on my journey.

**ברוך אתה יי אלהינו מלך העולם הטוב והמאטיב.**

For you, for reading this. Thanks.

# CHAPTER 1

# INTRODUCTION

Motion planning problems are well-known and well-studied in robotics, but motion plans rarely take computational restrictions into consideration when composing plans for robots or other autonomous devices. A computational restriction is a constraint on the system relating to its ability to run computation and the effect it has on the state, such as a constraint on battery life or maximum temperature. Computational restrictions are important for solving motion planning problems, as even a plan that obeys all physical constraints, such as not colliding with obstacles and moving in a physically possible fashion, may still be impossible to execute if it places undue stress on the computational devices during the process. While naïve solutions are possible, they may dismiss valid plans due to being overly conservative in avoiding certain motions.

We demonstrate the following: *By using Petri nets to model the requirements and effects of computation on mobile robots, we can find shorter solutions to motion planning problems and avoid new types of obstacles such as overheating the device and dynamic velocity constraints.*

As an example, consider a robot looking for civilians in a burning building. The robot must determine a plan to navigate each room without colliding with obstacles or running out of battery. Different parts of each room are different temperatures, and the high ambient temperature of the location may result in the robot's processing units being throttled due to overheating when too close to heat sources. As such throttling is unacceptable in such an environment due to it being unpredictable in its degree of slowdown of computation, a plan to navigate the building must take into account proximity to heat sources. A plan that draws closer to a heat source should shift computation to lower-heat-generating processing units even at the cost of computation speed, until such a point that the computation becomes too slow and the plan is rejected in favor of one which avoids the heat source. Naïvely avoiding any areas near heat sources is sufficient to avoid overheating, but may disregard much faster solutions that are near such areas briefly.

Another case is the self driving car and constraints on stopping. Vehicles have a maximum deceleration and finite friction with the environment, which are dependent upon the velocity of the vehicle, but there are also constraints upon how quickly the computation must be completed. Even if there is adequate time to halt the vehicle before colliding with an obstacle, if the image recognition cannot recognize an obstacle in the road quickly enough to begin braking, the obstacle may still be hit. As the velocity of the vehicle increases, not only does the stopping distance required increase, but so does the speed of this computation and the distance at which it may be performed. This is exacerbated by conditions such as nighttime and

heavy fog, where image recognition may be further slowed. As the velocity increases, greater pressure is put upon time completion of the computation, up until certain velocities under certain circumstances will be barred entirely. A naïve maximum on velocity does not sufficiently capture the nuance of the situation. We must be able to consider these dynamic requirements on our plans.

Further, consider that modern computing systems are typically heterogeneous, containing multiple processing units with distinct designs and performance characteristics. A common system-on-chip (SoC) design will include a central processing unit (CPU) often with multiple cores, a graphics processing unit (GPU), and a deep learning accelerator (DLA).

When a mobile robot is executing plans, it must also complete various computational tasks in order to update its state and avoid colliding with obstacles [1]. Examples of computational tasks are object detection, updating the internal state representation, and computing corrections to the path if the motion plan could not be followed precisely (such as if an unknown obstacle was present). A given computational task may be able to be executed on more than one of the processing units, though the requirements for time and power may vary. It is useful to use slower, but more energy-efficient processing units in cases where energy is limited or heat generation should be restricted — or, contrarily, where computation speed is of the essence and power not a concern, to use faster processing units when necessary. We see from [2] that efficient energy usage is important for the robot's performance. Current systems often make worst-case assumptions about energy usage, heat, and time required to complete tasks [1], which disallows many plans which may be faster or more energy-efficient. It is therefore desirable to have a motion planning system which takes into account different possible computation schedules and incorporates their effect, as well as the effects of the environment, into the motion planning. We present such a system below.

Our contributions are as follows:

- We create a new Petri net-based system for determining computational feasibility of a plan.
- We show that this may be used in a planning system to generate more sophisticated and efficient solutions to motion planning problems, and create a new system to do so.
- We demonstrate our work on several problems. We also discuss the early elements of using Petri nets for interprocessor interaction in the form of memory contention.

# CHAPTER 2

# RELATED WORK

Our work incorporates elements from motion planning, scheduling on heterogeneous computing systems, and the formal language model known as the Petri net.

## 2.1 Motion Planning

Motion planning is a well-studied problem, with many varied algorithms for different scenarios or requirements [3, 4]. One of the most influential papers on randomized motion planning is Rapidly-Exploring Random Trees (RRTs) [5, 6]. RRTs are a sampling-based approach to exploring the space [7], of which many variants such as RRT-Connect [8].

The RRT is an example of a geometric sampling-based planning problem. The RRT grows a tree from an initial location to the goal state. Nearby locations to current points on the tree are sampled, and a path found from the tree to the new point. If the path is found to collide with obstacles, the new point is discarded, and it is otherwise added to the tree. As the RRT is probabilistically complete [5], it will find a solution if one exists with probability 1 in the limit as time increases.

Many variations of the RRT exist. RRT-Connect samples starting from both the start and goal states of the space in order to improve the speed at which a solution is found. An example of an RRT may be seen in Figure 2.1. Other versions of RRTs include Anytime RRTs [9], the RRT* [10, 11], the Rapidly-Exploring Random Belief tree for planning under uncertainty [12], the sparse stable RRT [13], and the ikRRT and BkRRT [14].

RRTs have been used for varied contexts, including hybrid planning [15], replanning mid-execution [16], and even molecular dissassembly [17]. Related methods that are not RRT-based include KPIECE [18], the Fast-Marching Tree [19], and PDST-EXPLORE [20]. An example of an RRT path may be seen in Figure 2.1, sourced from [21] (see appendix B for copyright).

Especially important to our work is the control-based RRT (CRRT) [6], on which our algorithm is based. The Sparse Stable RRT [13] is a variant of the CRRT with asymptotic optimality, which could be used in tandem with our presentation. Asymptotic optimality is the property that the plan will converge to the optimal (that is, shortest) path with probability 1 in the limit of time.

Control-space planning is a variation on geometric-space planning which does not allow the selection of arbitrary nearby points. This is useful in cases where the constraints are non-holonomic, so that paths between points may be complicated to compute. An example of a control is selecting the robot's velocity,

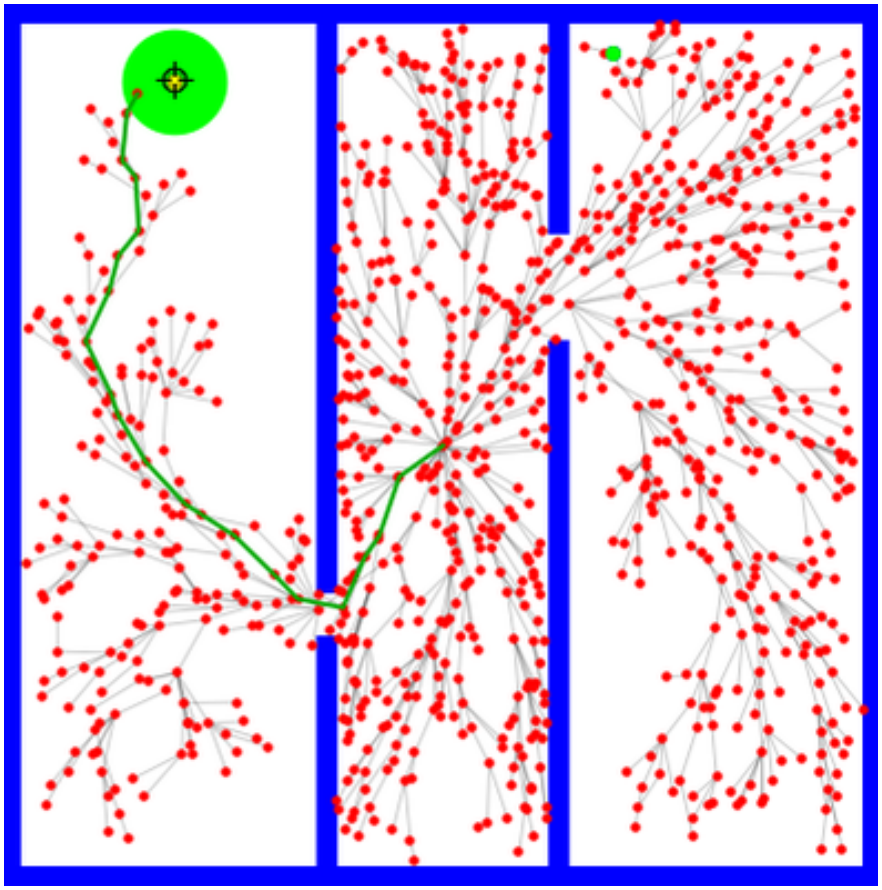

Figure 2.1 A Rapidly-Exploring Random Tree (RRT) simulation demonstrating how a path that avoids obstacles is found.

rather than its exact position. In our case, the computational constraints may not be directly controlled or moved to, and so are best handled with control-space planning.

The Control RRT is an example of a control-space planning algorithm and is a variation on the geometric RRT described above. Instead of randomly selecting nearby locations in the state space and determining the path to get there from the current tree, an external control is applied to a given state space location, causing an indirect change to the state. The control is then applied through a differential equation or other propagator function, which determines the new position in the state space. Similar to the geometric RRT, the new point is discarded if obstacles were collided with along the path and otherwise added to the tree.

Improvements to navigational algorithms such as the RRT, or alternative versions of it, could be used to solve the computational constraints problem. This could be done by biasing the RRT in the direction of desirable computation plans or creating an entire new RRT structure. Our approach is extensible and may be used in tandem with any of the above RRTs as desired by the user and the problem in question.

Sampling-Placed Motion Planning with Temporal Goals [22] uses temporal logic to discuss motion plans with temporal goals, such as a condition being required "at all times", "eventually", or "followed by" another. This is accomplished by creating a discrete approximation to the system, making high-level plans with temporal logic, and using traditional motion planning techniques to accomplish those plans. Linear temporal logic could also be used to encode our computational constraints, and so this presents an alternative approach for our problem.

Our system is used only for simple navigational goals, though the algorithm presented could be used in more complex scenarios or as the low-level motion planning within the larger temporal framework as long as the communication back to the discrete layer were implemented as part of the search algorithm. In addition, the motion planning done by [22] is geometric, making the assumption that position can be specified precisely, whereas our work uses control planning instead. Expanding our system to more sophisticated goals such as those shown is a promising direction for future research.

In the case that multiple goals are anticipated, rather than just one, another method is the probabilistic roadmap. The probabilistic roadmap does additional preprocessing so that replanning may be done, as well as changing goals partway through execution or solving multiple goals in an initially unknown order [23]. Much research has been done on probabilistic roadmaps and their various algorithms, including [24–29, 29–32]. Combining our work with an underlying probabilistic roadmap, rather than assuming a singular goal, is another possible direction for future work.

### 2.2 Hardware Performance Characteristics

When a computing system is running, it is affected both by its environment and the tasks being executed. Keeping track of parameters such as the amount of power remaining in the system, how quickly computation can complete in certain cases, and other permformance characteristics is thus important to create an accurate model. The Sky is Not the Limit [33] uses a form of the roofline model, common in high-performance computing contexts [34]. The roofline model is a type of log-log scale plot which shows a straight-line increase followed by a flat line at a "critical point" where the intensity of computation no longer leads to improved operations per time unit. These are frequently used as a diagnostic tool, determining which of various performance improvements will be significant. The Sky is Not the Limit uses such a model to show how operational intensity affects computation speed in the context of a flying drone and its maximum acceleration. Similar to our work, The Sky is Not the Limit determines the maximum velocity of a device according to computational considerations of a heterogeneous computing unit. However, its primary goal is diagnostic, and it does not attempt to schedule tasks in order to maximize this performance. Nevertheless, it demonstrates useful results and its modeling system can be used for our

work, using their system's results to determine appropriate parameters of our model.

Systems that do incorporate scheduling include [35–41]. Each uses a unique modeling system for optimal scheduling, but none are used explicitly in a motion planning context or take into account our considerations.

The literature of systems which dynamically consider computational constraints on a mobile robot is sparse. One notable example is RoboRun: A Robot Runtime to Exploit Spatial is Heterogeneity [1], which dynamically chooses maximum permissible velocities depending on system constraints and the current position and conditions of the environment. RoboRun takes into account the proximity of the mobile robot to obstacles and the current velocity, and adjusts the precision of its environment model as well as the portion of the environment being modeled.

This is done mid-execution, rather than during planning as in the below presentation, and does not possess formal guarantees. The work also does not take into account dynamic heat constraints, but only limitations on velocity. Unlike our system, RoboRun updates its constraints throughout the execution in a continuous manner, and allows for multiple versions of the required computations. Compared to RoboRun, our system is able to create motion plans before exection, as well as handle more differential constraints rather than only velocity. However, we do not have the ability to adjust computation on the fly. Allowing for multiple forms of possible computation is a direction for future work.

### 2.3 Petri Net Related Work

Petri nets are a type of discrete event system and are treated in depth in [42]. They are discussed in more detail in the background section of this work (subsection 3.1). Petri nets are a system that models resources as tokens in places, and transitions which consume or create more resources. They are a useful and compact way to represent concurrent and parallel systems, which is the manner they are used in our work. Many variants exist, including the timed Petri net in which transitions take a deterministic amount of time to fire [43].

First-Order Hybrid Petri Nets: A Model of Optimization and Control [44] formalizes a hybrid Petri net model, which allows for continuous transitions in addition to the discrete ones used in the following presentation, and solving their systems with sensitivity analysis.

The context under which their model is tested, application and manufacturing systems, is different from our use case, though we believe that our system could be used in other domains with some modifications. In addition, the method by which the Petri net is solved is unlike ours, which uses a discretized approximation to the underlying continuous structure. Using their system or a modification of it in our problem domain could lead to significantly improved performance of the schedule generation in our work

(see section 5), opening the door for other possible optimizations and techniques.

One system which does use Petri nets in a similar context is Petri Net Models for Single Processor Real-time Scheduling [45], which does task scheduling with a Petri net model on single-processor systems. While it takes into account some aspects our current model does not, such as shared resources across several tasks, it is only usable for single-core systems. It would therefore be insufficient for a modern heterogeneous computing system with multiple processing units with unique performance characteristics.

# CHAPTER 3

# BACKGROUND

Our work is an augmentation of a motion planning algorithm with a Petri net for scheduling, which we treat as a constraint solving problem. We provide a brief overview of these topics here.

## 3.1 Petri Nets

Petri nets are a formal language on infinite-length strings [42], useful for describing systems with discrete states. They are particularly useful for describing parallel or concurrent systems, as they can easily represent shared resources or two independent systems working simultaneously. They are used in our system to represent computational tasks.

A Petri net is a directed graph composed of two types of vertices: *places* and *transitions* [42]. Places have edges only to transitions, and transitions have edges only to places, and the graph is thus bipartite. A place contains a nonnegative number of *tokens*. A transition may *fire* or activate to decrease the token count at all places connected via incoming edges and increase the token count at all places connected via outgoing edges. Edges have weights, which denote how many tokens are removed or placed when a transition fires. The tokens that are required for a transition to occur are known as preconditions and the tokens placed after a transition fires are known as postconditions.

A *marking* is defined as the current state of the Petri net. A marking $m_i$ permits another marking $m_j$ if there exist a set of transitions that may be applied to $m_i$ that results in $m_j$. We denote this as $m_i \to m_j$. More than one transition may be applied to a marking to reach another by this definition. A given marking may permit multiple other markings.

A Petri net has a starting marking, referring to the marking at the beginning of its execution. A *trace* is a sequence of markings $m_0, \ldots, m_n$, such that $m_0$ is the starting marking, $m_n$ is a marking, and for each pair $m_i, m_{i+1}, m_i \to m_{i+1}$. A trace is defined by which transitions fire between each pair of markings.

In our work, we use Petri nets that vary from the typical model in two main ways. The first is the use of real-valued tokens, enabling places to take on any nonnegative value, rather than only nonnegative integers. This allows a generality which is useful in our model. An example may be seen in Figure 3.1. The second difference in our formulation is the introduction of a goal marking. We allow Petri nets to have one or more goal markings, defined by a specific range of tokens on each place. These may require places to have an exact number of tokens, a range of values, or an arbitrary value, that is, the range $(-\infty, \infty)$. Execution ends when a goal marking is reached.

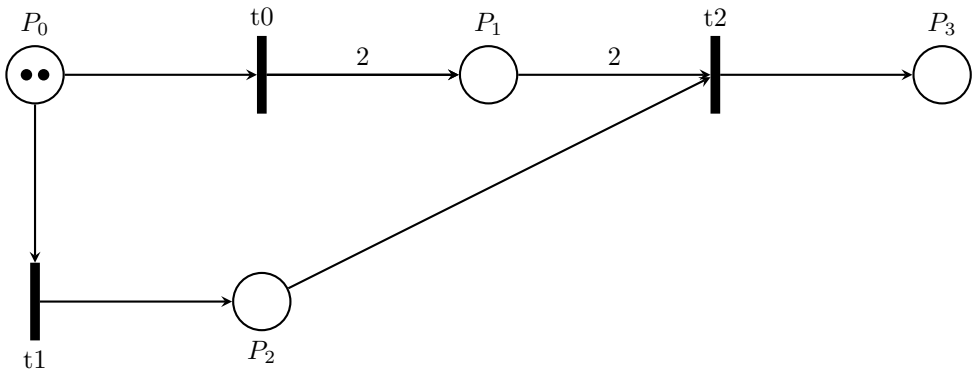


Figure 3.1 A basic Petri net. The circles are places and the bars are transitions. The starting marking has two token at P0. All unmarked vertices have a weight of 1. If a goal marking is to have one or more tokens at $P_3$, then a trace for this Petri net is to fire transition $T_1$ followed by $T_0$, followed by $T_2$.

### 3.2 Constraint Solvers and Constraint-based Planning

Constraint solving is a common approach for finding satisfying parameter values in robot planning in order to accomplish some particular goal. Our work uses constraint solvers to find traces of Petri nets.

A constraint solver is a program to find values for all parameters so that various constraints are satisfied. Examples include the famous SAT problem, subgraph modulo theory solvers, and mixed integer linear programming (MILP). Various powerful constraint solvers exist that can handle linear and quadratic constraints, such as Gurobi [46]. The system is organized in terms of actions, the preconditions and postconditions of those actions and how they affect the world space, the starting and goal configurations, and other constraints upon what actions may be taken under which conditions.

Constraint-based planning uses constraint solvers in order to create task plans for a system to execute. In this case parameters represent actions taken at certain times and their effect on the world. For instance, we may have one parameter describing a robot picking up an object at some given time, and others to describe the state of the world (such as the cup being on the ground or picked up by the robot).

As we do not know before beginning execution how many actions may be required to complete the plan, it is typical to describe the actions and world state without specifying the number of steps, and unroll the actions to increasingly large time steps. For each possible action and parameter, and over each time step from 0 to $n$, the problem state is unrolled over increasingly long time horizons. A maximum time horizon is selected, after which the problem is deemed unsolvable (or at least taking too long to find a solution).

# CHAPTER 4
# PROBLEM DEFINITION

Before a robot moves through a space, it determines a path through the state space to follow, known as a plan. We must create a motion plan that avoids both physical obstacles and avoids violating any computational constraints.

Computation must be *scheduled*. This refers to determining which processing units (henceforth PU, i.e. CPU, GPU) compute each task, at what times, and in what order. Scheduling tasks across heterogeneous systems is a relatively new problem, and considerations extend beyond those seen in operating systems such as fairness and response time. Further, we also have extra information to leverage that a typical operating system does not, due to the predictability of the tasks to run and their computational requirements.

Not every schedule may be performed at some arbitrary time, as there exist constraints that render some schedules impossible due to overheating the robot, running out of battery, or other such issues. We must, then, determine a schedule that is *valid* for each motion. A motion for which no schedule can be found is considered to be in the obstacle region. For instance, the mobile robot may not be able to move next to a heat source if the mobile robot is already quite hot, as it may overheat, but may be able to do so briefly if the core is currently cool, thus resulting in a more nuanced type of obstacle region.

Our problem is the tuple $\Omega = \{C, B, \tau, \lambda, \mathcal{C}, \kappa, q_0, q_1, \varepsilon\}$, where:

- $C$ is a set of constraints on the system, such as maximum permissible core heat or starting energy. These are of the form $c_i \in [x_{\min}, x_{\max}]$.
- $B$ is a finite set of $\beta$ PUs, $P_0, \ldots, P_\beta$, which the robot possesses (henceforth chip, core). Each $P_i$, when operating, has some effect on the robot's motion in the state space.
- $\tau$ is a finite set of $\gamma$ tasks $\tau_0, \ldots, \tau_\gamma$, along with the length of time each task $\tau_i$ requires on a given PU.Note that some PUs may be more suited to certain tasks than others, and so tasks may not be uniformly slower or faster on one PU over another.
- $\mathcal{C}$ is a state space through which the robot is to navigate, including a free region $\mathcal{C}_f$and an obstacle region $\mathcal{C}_o$ as defined in [23]. The state space includes the dimensions in the physical space in $\mathcal{R}^n$, as well as additional dimensions corresponding to $C$ such as the robot's remaining battery capacity. While parameters such as battery capacity do not correspond to physical space, it is a dimension in the state space. Henceforth, position and location will be used to refer to the location in the state space, so that the same physical space but different parameters lead to different positions, unless

otherwise specified. Locations where it is impossible to create an assignment of tasks to processing units and complete all computation within constraints is considered to be within $\mathcal{C}_o$.

- $\kappa$ is a control space, representing controls that may be applied to the mobile robot.
- $\lambda : \mathcal{C} \times \kappa \mapsto \mathcal{C}$ maps from a position in the state space and a control applied to a new state. It is henceforth called the *propagator* or *propagator function*.
- $q_0$ and $q_1$ are starting and goal states. $q_0$ is a single state, but $q_1$ may be a set.
- $\varepsilon$ is an error tolerance for goal proximity.

Given these inputs, the output is a continuous path $\sigma[0, 1] \in \mathcal{C}_{free}$ such that $\sigma_0 = q_0, \sigma_1 \in q_1, \forall_i \sigma_i \in \mathcal{C}_f$. Intuitively, this refers to a path that traverses from the start state to the goal state, and never enters the obstacle region — in this case, both physical and computational obstacles.

CHAPTER 5

METHOD

We extend the state space so that each of the computational constraints is a new dimension. A motion is now defined not only in terms of the device's physical movement through space, but also in terms of the physical effects of computation. The state space must grow to account for this and the obstacle region gains new obstacles. Our work integrates control-space motion planning with constraint-based scheduling of heterogeneous computing, and allows us to map discovered schedules to new locations in the state space. We incorporate a scheduling step into the propagator of the control space planner. Figure 5.1 shows the general structure of our program.

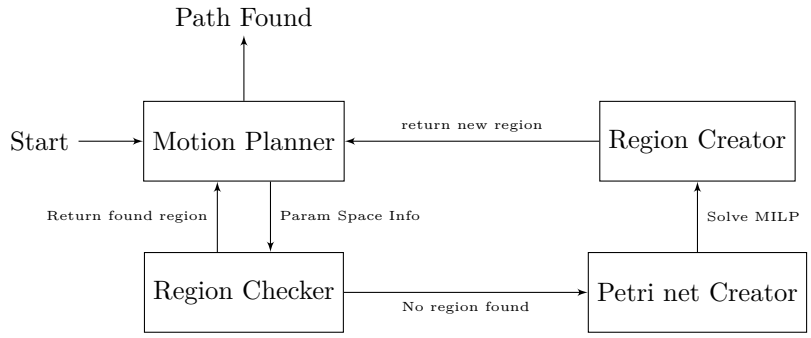


Figure 5.1 A block diagram demonstration of the method layout. The motion planner calls the region checker, which determines whether a currently existing region exists which satisfies the current constraints. If such a region exists, control is returned to the motion planner. If not, the Petri net creator/translator is called, which creates the relevant Petri net for the simulation and translates it to a satisfiability problem. The satisfiability problem is then solved, and the answer is used to create a new region.

### 5.1 Scheduling

We use Petri nets to determine valid schedules. We create a model of the computation systems the robot possesses, with its various PUs with different performance characteristics, and the tasks that must be completed. A trace of the Petri net corresponds to a valid schedule where all tasks were completed within the time limit and without constraint violation, as it describes when to execute each task and on which PU. If no trace can be found, then we consider the motion to be computationally infeasible. We use the same Petri net for all motions, but change the starting marking depending on the state and control spaces.

The Petri net is composed primarily of *task subunits*, along with a few additional places and transitions. A task subunit represents a task throughout its execution, from being not yet started to running to completed. We have one task subunut for each task. Additional places represent each PU. We also have

some *resource places*. A task subunit represents a computational task, which at any given moment is either not started, completed, or currently being executed by exactly one PU. If the core has $n$ PUs, then each task subunit contains $n + 2$ places. One place corresponds to a task not having been started, one for its having been completed, and $n$ for it being partway through its computation on one PU. Each PU has a place denoting whether it is free, and all such places contain one token in the starting marking. Only free PUs can begin to execute a task, and the PU is not freed until the task is finished. Transitions are added so that when a task is not started and a given PU is free, it may begin executing the task. When the PU's active-time place is sufficiently full, another transition frees the PU and sets the task to complete. A goal marking is any in which all task-completion places contain a token.

Figure 5.2 shows a task subunit for some task $i$ for two PUs, along with the PU places. The tokens shown are in the starting marking. Exactly one of $t0, t1$ will fire, resulting in task $i$ being executed on that PU. After some time and changes to the resource places (transitions relating to resource places are not shown here), one of $t_2, t_3$ fires to mark task $i$ as completed and the PU place is returned its token to complete another task.

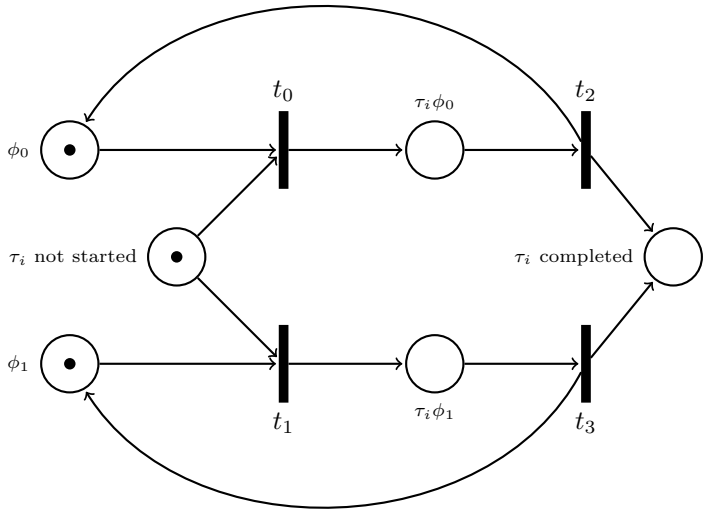


Figure 5.2 An example task subunit for a 2-PU case. Tasks are denoted with $\tau$ and processing units with $\phi$. $\tau_i\phi_j$ represents task $i$ executing on processing unit $j$.

The resource places are used to track the various parameters. A PU draws from different parameters while executing, and possibly replaces them upon completion of a task or while execution on a per-parameter basis (this depends on the nature of what the parameter represents, and so the precise behavior of each parameter is a modeling question).

If a valid schedule is found through the Petri net, then this schedule may be used in the computation while satisfying the constraints. We then use the equations of the previous section to maximize the size of this region for future use.

Schedules impact resource use, and therefore position in the state space. For instance, different schedules may use different amounts of energy. Thus, selecting appropriate schedules is important for performance of the plan. To find the optimal plan, we wish to return to points which already have schedules and search for additional ones, so that the space is explored more fully. We leave this to future work.

A trace of the Petri net corresponds to a scheduling of tasks on PUs. As the trace concluded in a goal marking, the schedule ensures that all tasks were completed and no constraints were violated. The schedule computed is used to determine the new value of parameters, and therefore new the position in the state space, when a control is applied. We therefore compute traces as part of propagating the control, and use the resultant schedule to determine the new state upon taking this action. If no schedule could be found, the state is set to an impossible value that will guarantee rejection.

The goal of our constraint satisfaction problem is to find a set of times to begin execution of tasks on different processing units so that all computation is concluded within the time limit. The decision variables include actions that may be taken at a given time step, and the value of various parameters (which may be real numbers, integers, or Booleans). The constraints include that the status of a given item only changes if an event occurs to change it, and define the way that a given action affects the world space. In modeling a Petri net, this means that the number of tokens at a given place can only change between time steps $j, j+1$ if a related transition fires, and it must change by the amount defined by the transition or transitions that fire.

We have various constraints on the fluents, so that they reflect the execution of a Petri net rather than assuming arbitrary values. These include precondition constraints, transition exclusivity constraints, and flow constraints. Places' values are constrained so that they reflect the firing of transitions over time and transitions are constrained to represent firing.

Formally, places are mapped to fluents of real type:

$$\varphi_i : \mathbb{R}$$

We denote $\varphi_i$ at time step $j$ as $\varphi_{ij}$.

If a transition $\xi$ requires, in order to fire at time step $j$, places $\varphi_0, \ldots, \varphi_k$ to have token counts at least $\theta_0, \ldots, \theta_k$, we generate the following constraint, where $\xi_j$ refers to $xi_j$ activating at time step $j$:

$$(\neg \xi_j) \vee (\forall i \in \{0, 1, \ldots, k\} \, (\varphi_{ij} \geq \theta_i))$$

We may equivalently write this as

$$\xi_j \implies (\forall i \in \{0, 1, \ldots, k\} (\varphi_{ij} \geq \theta_i))$$

In most constraint satisfaction contexts, only one action may occur at a single time step. However, as more than one transition may fire in a Petri net at once, we generally allow multiple transitions to fire at once. However, we do have this restriction for places which may only have token counts of 0 or 1. This is to avoid cases where the same task might be launched by both processing units simultaneously. If at time step $j$, both PUs are free and task $i$ is not started, it would be possible to fire both transitions at once without this restriction. With task_$i$_not_started being set to 0 at time step $j + 1$.

As such, for a place with these restrictions, which is affected by transitions $\rho_0, \ldots, \rho_k$, we have the following constraint for each $\rho_i$ ensuring it is affected by only one event. Here, $\rho_{ij}$ refers to the firing of transition $\rho_i$ at time step $j$.

$$\rho_{ij} \implies (\forall rho_k \neq \rho_i(\neg\rho_{kj}))$$

All places $\varphi_i$ have their new values at time $j + 1$ as the sum of all affecting transitions at time $j$. This is a flow constraint, ensuring that resources are not added to the Petri net without a relevant transition. If the possible transitions affecting $\varphi_i$ are $\eta_0, \ldots, \eta_k$, which have the value of 1 if the transition fires and 0 otherwise, and the transition firing affects the quantity of tokens at $\varphi_i$ by $\zeta_0, \ldots, \zeta_k$, we have the following constraint:

$$\varphi_{i(j+1)} = \varphi_i + \eta_{0j}\zeta_0 + \cdots + \eta_{ij}\zeta_j$$

When finding a new schedule, we set the starting marking of the Petri net and translate the Petri net into a mixed integer linear problem (MILP). The solution to the MILP is a trace of the Petri net, from which we find a schedule and its impact on the state space. We may solve the MILP with any constraint satisfaction program which supports problems of this type.

### 5.2 Regions for Schedule Validity

A naïve approach would compute a valid schedule for each call to the propagator function, and such an approach would indeed result in a working algorithm. As computing a schedule is very computationally intensive, doing so would mean even simple motion plans would take hours or days to complete. Instead, upon finding a new schedule, we determine the maximum set of positions in the state space and control space for which the schedule remains valid, henceforth *region*, and cache the schedule along with this region.

In order to determine if the schedule exists, we need to know certain aspects of the current state and control, but not in its entirety. For instance, in order to calculate the new temperature after executing a specific motion, it is necessary to know the ambient temperature, but not the exact position. As such, we

maintain a *parameter space*, a $q < n$-dimensional space corresponding to only that information which is necessary to compute a schedule. There exists a surjective mapping from the state space and control space to the parameter space.

We write the constraints in terms of finite resources, and consider a schedule valid if it does not require more of any parameter than is present. In the case that a parameter upper bound is required, it may be trivially added to the constraints. No upper bound was required in our work.

Upon finding a valid schedule, we wish to find a region containing the current point in which that schedule is valid. For any future motions, we check if the motion's position in the parameter space is in a region with a valid schedule. If it is, we reuse the schedule, thus avoiding any recomputation.

In the case that multiple regions overlap (that is, more than one discovered schedule is valid), we greedily select the first schedule we find when enumerating each region. A method to select the optimal of several schedules is left to future work.

Similarly, if a particular point in the configuration space has no valid schedules, it follows that any point for which we have even lower parameter values, is lower for all resources, this point will also have no valid schedules. We may thus construct a different type of region, called a *reject region*, within which no schedule exists.

We constrain our system. Each parameter $x_i$ has a constraint of the form $x_i \geq \alpha_i$.

The larger the region, the more cases in which we may reuse the schedule. Since we want to reuse schedules as much as possible, we want the regions to be as big as they can be. As each parameter is independent of each other, and when modeling as finite resources we use the constraint $x_i \geq \alpha_i$, the region therefore is trivially of the form

$$x_i \in (\alpha_i, \infty)$$

for all $x_i$. More complicated formulations such as optimization problems could be used for more complicated regions, but it is unneeded in our approach.

Note that, while the constraints must be in the inequality form seen
in (5.2Regions for Schedule Validitysubsection.5.2), this limit is only for maximizing the size of the schedule and is not reflective of how the parameter behaves in the Petri net. They may be anything from a finite resource that is used once and cannot be restored, to a resource that returns over time, to a complicated model.

In the case of a reject region from an evaluated point $p = p_1, p_2, \ldots, p_q$, the region is trivially the region defined as all points $z$ such that each coordinate $z_i \leq p_i$, which is the region defined by

$$x_i \in (0, p_i)$$

for each $x_i$.

### 5.3 Motion Planning

We use control-based planning to explore the state space. The CRRT (see subsection 2.1) is used for its ability to effectively explore a space with controls while maintaining the simplicity of the RRT. On each iteration of the algorithm, new point is added to the tree by applying a control. We map the current state space and control space to the parameter space and check the region list. If we are in a region with a known schedule, we use that schedule. If we are in a reject region, we discard the motion. If neither case is true, we create the starting marking for the Petri net, attempt to find a trace, and add a new region to the region list according to our optimization problem. The new region is a standard region if a trace could be found, and a reject region otherwise.

Algorithm 1 describes our core algorithm. We use line 7 to check our known regions if the motion and location has a known schedule. If no region currently exists, on line 9 we set the Petri net's starting marking. We create and solve the resulting constraint satisfaction problem on line 10. If a schedule is found, we create the standard region on line 14, and if not, we create a reject region on line 12. As we just created the region, we can guarantee that it exists. Online 18, we determine whether the motion enters the obstacle region before adding the motion to the tree, including a check for whether the schedule was rejected or not.

```
1  t ← startingPosition ;
2  ρ ← ∅ ;
3  φ ← createPetriNetFromTaskFile();
4  g ← goalPosition() ;
5  while dist(t, g) < ε do
6  |  m ← generateNewMotion(t) ;
7  |  r ← findRegionContainingMotion(m) ;
8  |  if nil(r) then
9  |  |  setPNStartingTokens(φ, m) ;
10 |  |  s ← solveForSchedule(φ) ;
11 |  |  if nil(s) then
12 |  |  |  addRejectRegion(ρ, m)
13 |  |  else
14 |  |  |  addMaxSizeRegion(ρ, s)
15 |  |  end
16 |  |  r ← findRegionContainingMotion(m);
17 |  end
18 |  if motionValid(m, r, t) then
19 |  |  addMotionToTree(m, t)
20 |  end
21 end
```

**Algorithm 1:** Motion Planning with Computation Constrants

We use pre-existing methods to check for invalidity of our resultant position in the state space, such as the Flexible Collision Library [47]. Invalid points in the state space include those with zero or negative remaining resources, as well as points that collide with obstacles such as walls. With this system, we may explore the search space and find a path from the start to the goal state that avoids physical obstacles and satisfies all computational constraints.

# CHAPTER 6

# EXPERIMENTS

Our experiments cover several simulated environments, and are designed to test constraints which are applicable to a wide range of types of robot and use cases. The constraints which we consider include:

- temperature
- energy usage
- power usage
- computation time

As our target architecture, we use an Nvidia Jetson AGX Orin Developer Kit [48], henceforth Orin. It has an NVIDA Ampere architecture with 2048 CUDA cores and 64 tensor cores and a 12-core ARM-Cortex A78AE v8.2 64-bit CPU with a 3MB L2 cache and 6MB L3 cache, as well as two deep learning accelerators (DLAs) and a vision accelerator. The target robot is the Clearpath Robotics Jackal [49]. Motion planning was done with the Open Motion Planning Library (OMPL) [50] using Amino [51] and TMKit [52], with Z3 [53] as the constraint solver. The algorithm was implemented in a combination of C, C++, and Common Lisp, and executed with Steel Bank Common Lisp (SBCL) [54] and the Common Foreign Function Interface (CFFI) [55] in an Emacs execution environment.

## 6.1 Tasks

Our task load runs on the GPU and Deep Learning Accelerator(DLA). We run three tasks, namely the neural networks Resnet, MobileNet, and GoogLeNet. We use these to represent semantic segmentation, object detection, and short-term navigational/correctional decisions respectively. We discretize motion into segments of length 1/5 seconds, such that all tasks must complete in this time period, and run MobileNet and Resnet five times in sequence on the same PU, but not the more time-demanding GooLeNet. Due to implementation constraints, rounding to the next highest integer of the time unit required was necessary for our system. With this rounding, the GPU always completed all computation in one unit, with the DLA being equally as fast on ResNet and three and four times slower on MobileNet and GoogLeNet respectively. The power and heat requirement was 1/16 on the DLA compared to the GPU. The data may be seen in Table A.1.

### 6.2 Scenarios

Our scenarios include:

1. A mobile robot in a burning building. There is one heat source between the start and goal locations. The naïve solution is to avoid all areas with an ambient temperature greater than some threshold, avoiding the heat source entirely but possibly foregoing an otherwise useful solution. Our work determines the core temperature of the PUs throughout plan execution, enabling the robot to get potentially very close to the heat source as long as the core processor temperature does not exceed its threshold.

2. A robot with a short distance to travel, but minimal remaining energy. Our system is able to detect this and use slower, but less power-intensive PUs, without ever being explicitly instructed to do so.

3. A mobile robot at night with limited visibility and obstacles in the way. Given the reduced visibility, the maximum permissible velocity to avoid running into obstacles is much lower than the robot's maximum. Our system is able to determine the maximum allowable velocity while still having enough time to complete all computation, without having an arbitrary maximum velocity imposed on it by a user.

Our state space exists in $\mathbb{R}^n$ and includes two values for the $R^3$ coordinates $x, y, \theta$, as well as other information that varies depending on the specific constraints and their relevant values. This may be trivially expanded in various ways, such as using $R^3$ for problems involving a device that may move in a third dimension.

The regions, parameter space, and Petri nets have no awareness of or dependence upon the propagation algorithm. As such, the propagator may be customized on a per-problem or per-robot basis without affecting the Petri net. For our simulations, we use the Unicycle Model [23], which allows for moving forward and backward and turning, including turning in place. This model is accurate to the movement capabilities of our robot, which are differential drive platforms. We assume that instantaneous changes in velocity are possible for simplicity of the model, but more realistic models or models for different types of robots would require no changes to the Petri net.

### 6.3 Physical constraints for schedule validity

Here we discuss the various physical constraints which our experiments test. We provide a brief overview of the underlying physical nature of each constraint and provide the equation or equations used to determine it.

### 6.3.1 Temperature and Heat

the temperature and heat of our system depends on the heat of the device and the heat of the surrounding air. We use Newton's Law of Cooling, a simplified system for simple cooling in constant temperatures without considering radiation [56], defined below.

$$\dot{Q} = k\Delta T \tag{6.1}$$

The constant $k$ is related to the size of the exposed area of the chip and the materials the robot is composed of, and must be determined experimentally for each robot and device. $\Delta T$ is the difference between the system's current temperature and the surrounding environment. $pdotQ$ is the instantaneous rate of change of temperature of the object in question.

If our chip exceeds its critical temperature, it will begin to throttle. We found experimentally that for our computation device, unpredictable throttling occurred at a core temperature of $100°C$. We found that the system would switch between different throttling modes many times per second, so that it was hard to predict its exact computation speed when throttling. As such, performance was sufficiently hard to predict that it became untenable for real-time computing to allow any degree of throttling.

We used a heat generation unit to test the Orin's cooling capabilities. The setup may be seen in Figure 6.1. Our experiments showed that throttling occurred almost immediately after exposure to external temperatures of $70°C$, and so our naïve comparison system avoids these temperatures unequivocally. For our algorithm, when a computation schedule is selected, we determine by what degree that schedule will increase the core temperature, and use the above heat dissipation equation to determine the new temperature at the motion's end. A valid schedule avoids overheating the device, but if the device's temperature is very hot and the ambient temperature does not permit fast cooling, no schedules may exist so that the motion is rejected. As attested in [57], the Jackal robot itself is rated to (a surprisingly low!) $45°C$.

$$\text{Constraint Equation: } T < T_{\max}$$

To test this parameter, we used a simple hallway containing a fire in the middle, with a heat of $800°C$. We used two models — our computationally aware model, as well as a naïve models which treats any region with a temperature greater than $T_{\max}$ as part of the obstacle region with no sensitivity to core temperature. The second model was run with $T_{\max} = 100, 70, 45$, corresponding to the Orin's critical temperature limit, the point of our experiment where intense computation would lead to almost immediate throttling, and the Jackal's maximum rated safe operating temperature.

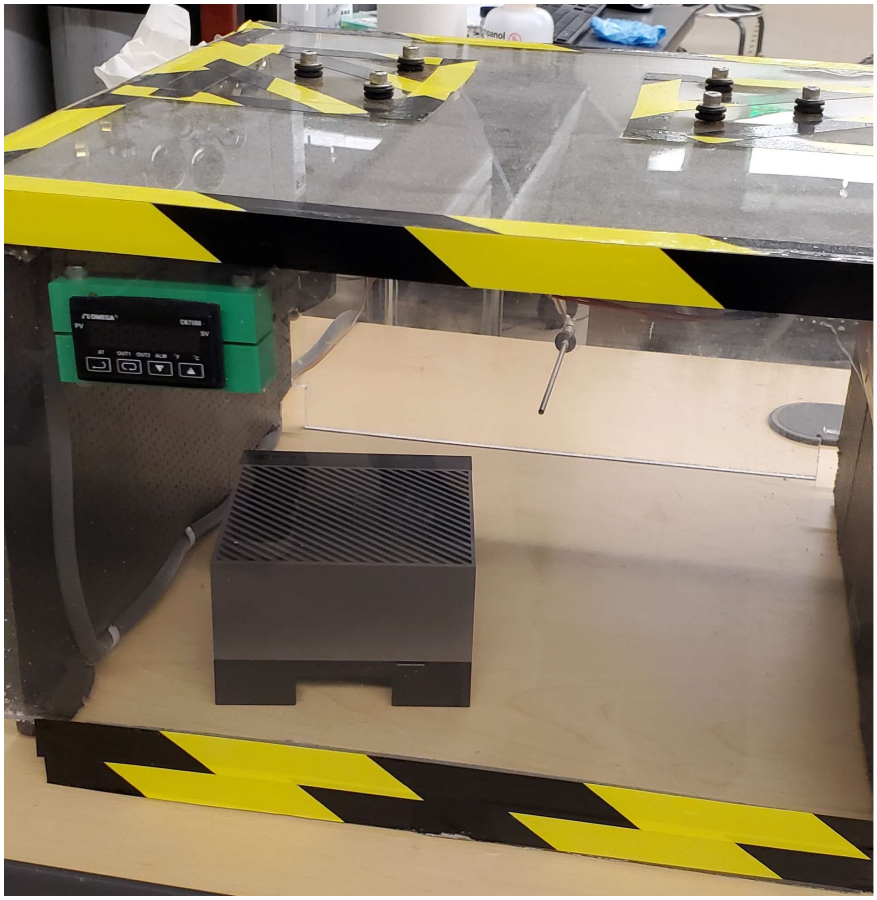

Figure 6.1 The heat generation unit used for testing performance of the Orin at different external temperatures.

### 6.3.2 Power Limits

At any given instant, the amount of power being used may not exceed a certain threshold due to physical limits on the battery and system in question. The maximum amount of power able to be used will be constant across all parts of the plan, as it is a physical constraint on the system's design. This constraint is constant across all locations in the parameter space. This constraint is of the form $r_p > s_p$, that is, the power drawn by the schedule $s$ must be less than what is available anywhere in the region $r$.

For a given motion and schedule, the power depends on various subparameters, including which PUs are active and for how long, and which actuators for the movement of the device are active and to what degree (i.e., propeller rotation speed). Let $p$ represent a PU and $a$ represent an actuator.

Constraint Equation: $r_p > s_p$,

$$r_p = \left( \sum_p p_{\text{ power draw }} \times p_{\text{ time active }} \right) + \left( \sum_a \int_{a_{ti}}^{a_{tf}} a_{\text{ power draw }} \right)$$

While this restriction is implemented in our system, we did not find that the robot, PUs, and battery in question approached a point where this limit became relevant.

### 6.3.3 Energy Capacity

Unlike other constraints we discuss, this follows through the entire motion plan and cannot be reversed. The system may not use the entirety of its energy capacity by the end of the plan execution. For one given motion, this decreases the energy by some amount, and so the requirement is that that amount of energy is available at that moment. Let $\mu$ represent a motion and $\mu_e$ the energy required by that motion. Let $\nu$ represent the total amount of energy stored in the system's power source, such as battery.

$$\text{Constraint Equation: } \sum_m \mu_e \leq \nu$$

We use our algorithm in a "long hallway with obstacles" scenario with increasingly low battery to see that it automatically assigns its tasks to slower, but lower-power PUs, so that it may complete its task without running out of energy.

### 6.3.4 Time and Velocity

The velocity during a motion influences how quickly the computation must complete. If moving more quickly, it takes longer to brake in case of an unforseen obstacle, and so it is required to finish computation more quickly so that emergency braking can be done faster (and thus avoid collision). In general, the time constraint is inversely proportional to velocity. Given a schedule which takes time $s_t$, we may thus find the maximum velocity that is permitted with this time requirement. $s_t \propto s_v^{-1}$, that is, the schedule's time is inversely proportional to the velocity. The constraint, written in terms of time, is $r_t > s_t$, that is, points where the time required to complete the motion is larger than time required for the schedule to complete.

Let $s_t, r_t$ be the time requirement of the schedule and the portion of the region respectively, $s_v$ be the velocity at the point the schedule is being executed, and $k \in \mathbb{R}$ be some constant.

$$\text{Constraint Equation: } r_t > s_t,$$

$$s_t = k s_v^{-1}$$

To test this parameter, we used the long hallway with obstacles in a nighttime environment, and increased its parameter $k$ accordingly to suggest that it must complete its computation faster, as many visual sensing devices work at reduced capacity at night. In cases where the system becomes unable to find the goal, we brought the goal closer to the start state and repeated. We see that this automatically reduces the device's maximum permissible velocity while still finding a solution, without any explicit programming to do so.

All tests were conducted with a 10 minute timeout. See Figure 6.2 for visual representations of the planning regions.

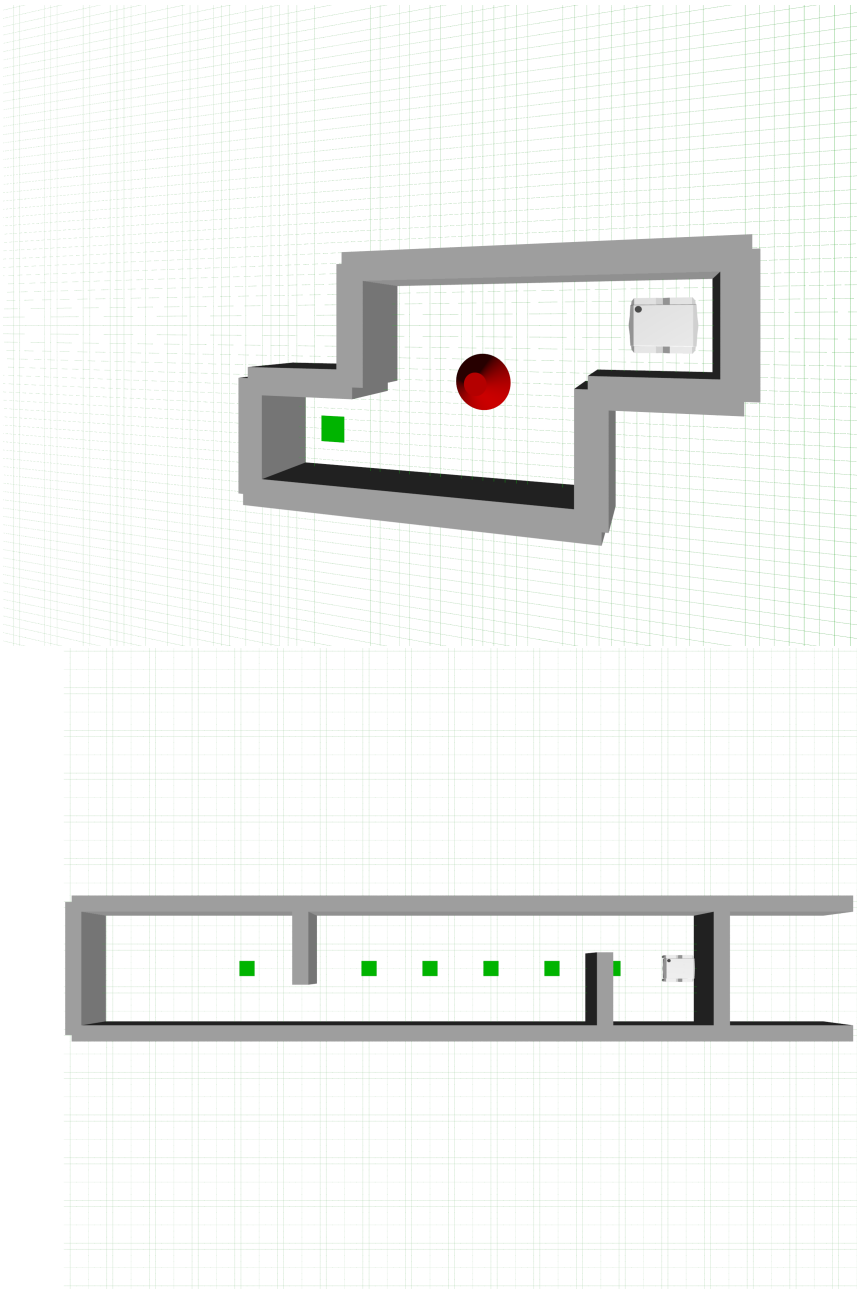

Figure 6.2 The test case areas. In Scene 1 (top), the red cone represents the heat source, and nearby parts of the scene are increasingly hot. In the second region (bottom), the different green squares are increasingly close to the start position to allow for changing the distance for the velocity test, but most tests enabled the robot to reach the final green square on the far left end of the space.

### 6.4 Results & Analysis

The plans and locations visited may be seen in Appendix A. We proceed through each of the three tests types, and end with some qualitative discussion of the plan results overall.

#### 6.4.1 Heat Test Results

We find that the plans generated by our system can pass very close to the heat source without concern. The resultant plans are relatively jagged and require more steps than those created by the naïve planner, but the solution is found consistently. Curiously, all three systems were able to pass through the heat source, even with the stringent $45°C$ restriction. This does not match with expectations, where systems with stronger constraints would have to give the fire a significantly wider berth than we observe. It is unclear whether this is a property of the system itself or if the parameters lead to this occurrence. Using the heat measurements from our system, however, it becomes clear that *all naïve systems would have overheated* . Our planning system avoids this by its design. The others pass too close to the heat source and spend too long there. Meanwhile, our system is much more deft at avoiding it, and when in the heat source moves through very quickly to avoid overheating. This shows that naïve plans can be risky, and in some clear sense "inferior" to our plans. The plans are shown in Figure 6.3.

#### 6.4.2 Energy Test Results

When energy is not a concern, the long hallway is solved quickly and without concern in a straightforward fashion. When the amount of energy available decreased substantially, the plans had a roughly 50% rate of failing to find a solution within the timeout (of the tests reported below, each was run twice for this reason). However, the number which were successful and the number which failed did *not* vary with the severity of the power restriction! That is to say, as long as a reasonable solution existed, whether the system was required to switch to power-saving schedules early or late in the plan did not make the solution any more difficult. This shows the flexibility of our system, in that while the problem does increase in difficulty when energy grows scarce, that difficulty increase is constant and so our planner has essentially arbitrary scalability in terms of the power constraint severity.

The reason this behavior is seen is because, as soon as the energy reaches the "critical value" where a schedule is no longer viable because it will use more power than is available, the next fastest schedule will always be found in the very next motion plan deterministically. This is a property of the region checker, which is not probabilistic in nature even if the motions themselves are. In this sense, it contrasts with trying to get through an arbitrarily narrow hallway where clearance approaches 0. In that case, the required motion to make it through the passage can only be found probabilistically, but here, the finding of

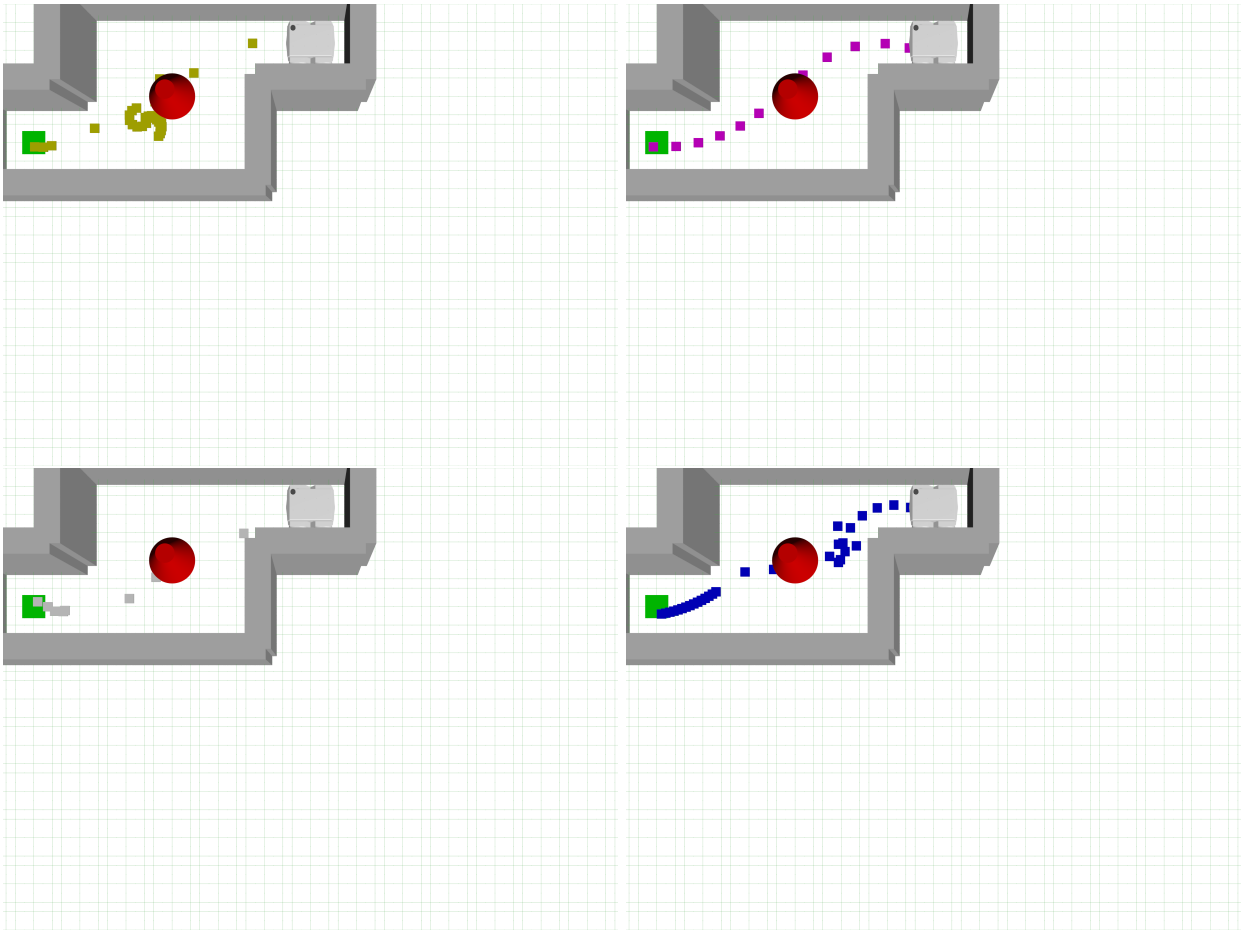

Figure 6.3 The heat avoidance plans are shown here. They are: 100 degree avoidance (top left), 70 degree avoidance (top right), 45 degree avoidance (bottom left), computation-aware (bottom right). Note that the computation-aware plan waits by the edge of the fire for an extended period of time before moving through in few motions.

the new schedule is guaranteed. Further, even if the motions initially selected will not lead to the goal, once the new schedule is found once it is usable by the motion planner for the entire remainder of the planning, so that future motions using it will be nearly as fast to plan as in a standard control motion planner.

Our planner is thus shown to be robust to even very severe power constraints, able to plan with similar levels of success with heavy power constraints as with light ones.

### 6.4.3 Velocity Test Results

The velocity tests show that our robot slows down smoothly, and planning difficulty increases only very slightly. The time required to plan generally did not increase, and failure was rare. The maximum range of even a 10% visibility decrease was noticeably shorter than that of full visibility, but afterwards there was a smooth decrease in maximum velocity without affecting planning time, with the goal consistently

remaining at the same distance. While various regions were created and then rejected, the system was typically able to proceed without difficulty. Planning became unduly difficult and ceased to result in successful plans even at a close range at an 80% visibility decrease, which resulted in a velocity near zero. The reached locations may be seen in Table 6.1.

Table 6.1 The visibility changes, maximum velocity, and obtained goal distance for the robot in the velocity tests.

| | | |
|---|---|---|
| 20 | 100 | 7.0 |
| 18 | 90 | 5.0 |
| 16 | 80 | 5.0 |
| 14 | 70 | 5.0 |
| 12 | 60 | 5.0 |
| 10 | 50 | 5.0 |
| 8 | 40 | 5.0 |
| 6 | 30 | 3.0 |
| 4 | 20 | 0.0 |
| 2 | 10 | 0.0 |

### 6.4.4 Current Limitations and Future Work

- In general, the plans were not meaningfully slower to compute using the Petri net system when only one region was required. This suggests that the overhead of the more complex algorithm is negligible. Despite this, plans were often, and somewhat randomly, slow to compute. OMPL's planner documentation notes that it will occasionally attempt to move directly toward the goal with a configurable bias parameter, but this is unused in the case of control plans. Its default CRRT behavior is to move in completely random motions with no bias, which led to us being somewhat "at the mercy of probability". The same plan could occasionally take a few seconds or a few minutes. A better/more specialized variant of the CRRT or a custom sampler could alleviate this.

- Another performance bottleneck was the region generation. Longer plans could often take over 60 seconds to compute, a troublesome amount when the timeout is merely ten times that. Compressing the PN representation, using a more powerful solver, a stronger computer to calculate the plans, or similar, would help, but ultimately this only delays the inevitable when the underlying problem is NP-hard, especially if we wish for the planner to keep track of more systems. An entirely different method for finding the PN's trace may be called for.

- When a motion is selected, a single schedule is utilized. This is usually greedily selected to be the fastest schedule, though if multiple previously discovered schedules overlap one is selected at random (in our implementation, the latest to have been found, though this is merely due to implementation

rather than a deliberate design choice). This, however, leads to certain cases where the algorithm may fail to find a path when one does exist, especially in the case of a power constraint.

Consider a long hallway, where the device has barely enough power to reach the end, but only if it conserves energy and uses the slowest, least power-intensive schedules possible at all points. Our algorithm will instead greedily select the fastest schedule available at all times until its power is so low that it is no longer even possible, and only then switch to a power-conserving schedule out of necessity. Such a plan will never be able to reach the goal, as too much power is wasted on fast computation at the start. At no point will the previously identified points be revisited to solve for additional schedules, thus locking in the robot to this flawed approach. To resolve this issue would involve recomputing previously found points to look for additional schedules to use, thus exploring in more directions than our planner currently does. However, to do so would require much more use of the constraint solver, bringing us back to the previously mentioned performance bottleneck.

# CHAPTER 7

# AN ALTERNATIVE PETRI NET SYSTEM: MEMORY CONTENTION

One notable omission from our simulation is consideration of memory contention. In a system on a chip like the Orin, there is only one memory for all PUs, meaning that two PUs requesting memory simultaneously will result in one, or both, being delayed. As such, when two PUs are active at once both will slow down by an amount that varies depending on the system and tasks in question. This is not accounted for in the simulation above, but unlike the other weaknesses discussed above, this one has a known solution. In addition to the Petri net system discussed above, we created another system that keeps track of a PU's external contention, referring to all memory requests not being made by that same PU and which will slow it down. It uses a specialized Petri net where certain places may take one of several enumerated modes based on the number of tokens in certain other places. This maps a continuous place to one of a few discrete values, while having more flexibility than simply integer tokens. For instance, an event may decrease the number of tokens in the continuous place, but by an amount not sufficient to change the mode of the enumerated place, or may change it by several at once. While this may be expressed in regular Petri nets as well, this proved a much simpler encoding that was also much faster in SMT solvers.

These modes change the speed at which computation proceeds. In our presented model, a computation takes the same amount of time on the same PU under all circumstances. In the memory contention model, each task on each PU has a known amount of data requested each time unit, and the amount that the PU obtains is the requested amount multiplied by an amount based on its external contention mode (typically less as the contention increases). As such, a task that proceeds quickly alone may be much slower when executed while other PUs also executing, such that it may be better to have only one active PU, or to begin one task on one PU partway through the execution of another. Models and analysis for slowdown due to memory contention are relatively uncommon, though there are some examples [58–61].

The model only concerns itself with memory contention, and does not keep track of In principle, these could be done together, and doing so would result in a much more detailed model. However, the substantial increase in the number of places and transitions in the Petri net would substantially decrease the speed of solution finding. Such a combination is contingent upon finding ways to improve the speed of schedule finding.

Difficulties in obtaining appropriate data means that this model has not been sufficiently tested as of this writing. It nevertheless remains a promising direction for future research.

# CHAPTER 8

# CONCLUSION

In this work, we have shown the utility of considering computational constraints when doing robot planning with heterogeneous computational devices. We show the use of Petri nets as a computation modeling and scheduling system, and demonstrated that our system can handle a range of constraints and scenarios without changing the system. The results are promising and indicate many directions for future work and improvements to the system.

There are several avenues for future work. We wish to add new considerations to the model, such as adding memory contention and simulating throttling at certain temperatures rather than simply avoiding those regions. Another possibility for expansion is searching with multiple regions rather than using just one, or having a better algorithm to decide a region's size (such as one using sensitivity analysis). We are also interested in improving the speed of the schedule generation so that more schedules (and more detailed schedules) can be generated, preferably with timed [62] or hybrid [44] Petri nets rather than the current discretized model.

Our key contribution is the introduction of a new system that incorporates constraint satisfaction for heterogeneous computation as part of a motion planning system. Our results are formally veriable and able to be extended to any kind of rapidly-exploring random tree.

REFERENCES


[1] Behzad Boroujerdian, Radhika Ghosal, Jonathan Cruz, Brian Plancher, and Vijay Janapa Reddi. Roborun: A robot runtime to exploit spatial heterogeneity. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 829–834. IEEE, 2021.

[2] Behzad Boroujerdian, Hasan Genc, Srivatsan Krishnan, Wenzhi Cui, Aleksandra Faust, and Vijay Janapa Reddi. Mavbench: Micro aerial vehicle benchmarking, 2019.

[3] Tomás Lozano-Pérez and Michael A Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.

[4] Oren Nechushtan, Barak Raveh, and Dan Halperin. Sampling-diagram automata: A tool for analyzing path quality in tree planners. In *Algorithmic Foundations of Robotics IX: Selected Contributions of the Ninth International Workshop on the Algorithmic Foundations of Robotics*, pages 285–301. Springer, 2011.

[5] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report TR-98-11, Computer Science Deptartment, Iowa State University, October 1998.

[6] S.M. LaValle and J.J. Kuffner. Randomized kinodynamic planning. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, volume 1, pages 473–479 vol.1, 1999. doi: 10.1109/ROBOT.1999.770022.

[7] Michael S Branicky, Michael M Curtiss, Joshua Levine, and Stuart Morgan. Sampling-based planning, control and verification of hybrid systems. *IEE Proceedings-Control Theory and Applications*, 153(5): 575–590, 2006.

[8] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 2, pages 995–1001. IEEE, 2000.

[9] Dave Ferguson and Anthony Stentz. Anytime rrts. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5369–5375. IEEE, 2006.

[10] Sertac Karaman, Matthew R Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. Anytime motion planning using the rrt. In *2011 IEEE international conference on robotics and automation*, pages 1478–1483. IEEE, 2011.

[11] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[12] Adam Bry and Nicholas Roy. Rapidly-exploring random belief trees for motion planning under uncertainty. In *2011 IEEE international conference on robotics and automation*, pages 723–730. IEEE, 2011.

[13] Yanbo Li, Zakary Littlefield, and Kostas E Bekris. Asymptotically optimal sampling-based kinodynamic planning. *The International Journal of Robotics Research*, 35(5):528–564, 2016.

[14] Chris Urmson and Reid Simmons. Approaches for heuristically biasing rrt growth. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, volume 2, pages 1178–1183. IEEE, 2003.

[15] Michael S Branicky, Michael M Curtiss, Joshua A Levine, and Stuart B Morgan. Rrts for nonlinear, discrete, and hybrid planning and control. In *42nd IEEE International Conference on Decision and Control (IEEE Cat. No. 03CH37475)*, volume 1, pages 657–663. IEEE, 2003.

[16] M Zucker, J Kuffner, and M Branicky. Multiple rrts for rapid replanning in dynamic environments. In *IEEE Conference on Robotics and Automation*, 2007.

[17] Juan Cortés, Léonard Jaillet, and Thierry Siméon. Molecular disassembly with rrt-like algorithms. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3301–3306. IEEE, 2007.

[18] Ioan A Sucan and Lydia E Kavraki. A sampling-based tree planner for systems with complex dynamics. *IEEE Transactions on Robotics*, 28(1):116–131, 2011.

[19] Lucas Janson, Edward Schmerling, Ashley Clark, and Marco Pavone. Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions. *The International journal of robotics research*, 34(7):883–921, 2015.

[20] Andrew M Ladd and Lydia E Kavraki. Fast tree-based exploration of state space for robots with dynamics. *Algorithmic Foundations of Robotics VI*, pages 297–312, 2005.

[21] Wolfram Alpha, Aaron T. Becker and Li Huang. Rapidly exploring random tree (rrt) and rrt*, 2018. URL https://demonstrations.wolfram.com/RapidlyExploringRandomTreeRRTAndRRT/. [Online; accessed May 9, 2023].

[22] Amit Bhatia, Lydia E. Kavraki, and Moshe Y. Vardi. Sampling-based motion planning with temporal goals. In *2010 IEEE International Conference on Robotics and Automation*, pages 2689–2696, 2010. doi: 10.1109/ROBOT.2010.5509503.

[23] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[24] David Hsu, J-C Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. In *Proceedings of international conference on robotics and automation*, volume 3, pages 2719–2726. IEEE, 1997.

[25] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *The International Journal of Robotics Research*, 21(3): 233–255, 2002.

[26] David Hsu, Jean-Claude Latombe, and Hanna Kurniawati. On the probabilistic foundations of probabilistic roadmap planning. In *Robotics Research: Results of the 12th International Symposium ISRR*, pages 83–97. Springer, 2007.

[27] Valérie Boor, Mark H Overmars, and A Frank Van Der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, volume 2, pages 1018–1023. IEEE, 1999.

[28] Michael S Branicky, Steven M LaValle, Kari Olson, and Libo Yang. Quasi-randomized path planning. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 2, pages 1481–1487. IEEE, 2001.

[29] Andrew Dobson and Kostas E Bekris. Sparse roadmap spanners for asymptotically near-optimal motion planning. *The International Journal of Robotics Research*, 33(1):18–47, 2014.

[30] Lucas Janson, Brian Ichter, and Marco Pavone. Deterministic sampling-based motion planning: Optimality, complexity, and performance. *The International Journal of Robotics Research*, 37(1): 46–61, 2018.

[31] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12 (4):566–580, 1996.

[32] Andreas Orthey and Marc Toussaint. Sparse multilevel roadmaps for high-dimensional robotic motion planning. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7851–7857. IEEE, 2021.

[33] Xingqin Lin, Vijaya Yajnanarayana, Siva D Muruganathan, Shiwei Gao, Henrik Asplund, Helka-Liina Maattanen, Mattias Bergstrom, Sebastian Euler, and Y-P Eric Wang. The sky is not the limit: Lte for unmanned aerial vehicles. *IEEE Communications Magazine*, 56(4):204–210, 2018.

[34] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers 5*, pages 129–148. Springer, 2015.

[35] Marvin Damschen, Frank Mueller, and Jörg Henkel. Co-scheduling on fused cpu-gpu architectures with shared last level caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2337–2347, 2018.

[36] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E Belviranli. Axonn: energy-aware execution of neural network inference on multi-accelerator heterogeneous socs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1069–1074, 2022.

[37] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. Deep learning inference parallelization on heterogeneous processors with tensorrt. *IEEE Embedded Systems Letters*, 14(1): 15–18, 2021.

[38] Duseok Kang, Jinwoo Oh, Jongwoo Choi, Youngmin Yi, and Soonhoi Ha. Scheduling of deep learning applications onto heterogeneous processors in an embedded device. *IEEE Access*, 8:43980–43991, 2020.

[39] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 209–221. Association for Computing Machinery New York, NY, USA, 2022.

[40] Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing. Blastnet: Exploiting duo-blocks for cross-processor real-time dnn inference. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, pages 91–105, 2022.

[41] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. Band: Coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 235–247, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391856. doi: 10.1145/3498361.3538948. URL https://doi.org/10.1145/3498361.3538948.

[42] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems, 2nd ed.* Springer, New York, NY, 2008. ISBN 978-0-387-68612-7. doi: https://doi.org/10.1007/978-0-387-68612-7.

[43] W.M. Zuberek. Timed petri nets definitions, properties, and applications. *Microelectronics Reliability*, 31(4):627–644, 1991. ISSN 0026-2714. doi: https://doi.org/10.1016/0026-2714(91)90007-T. URL https://www.sciencedirect.com/science/article/pii/002627149190007T.

[44] F. Balduzzi, A. Giua, and G. Menga. First-order hybrid petri nets: a model for optimization and control. *IEEE Transactions on Robotics and Automation*, 16(4):382–399, 2000. doi: 10.1109/70.864231.

[45] Martin Naedele. Petri net models for single processor real-time scheduling. *Citeseer*, 1998.

[46] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2021.

[47] Jia Pan, Sachin Chitta, and Dinesh Manocha. FCL: A general purpose library for collision and proximity queries, 2012.

[48] Nvidia. *Nvidia Jetson AGX Orin Developer Kit*. Nvidia.

[49] Clearpath Robotics. *Jackal(TM) Unmanned Ground Vehicle Technical Specifications*. Clearpath Robotics, .

[50] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.

[51] Neil T. Dantam. Robust and efficient forward, differential, and inverse kinematics using dual quaternions. volume 40, pages 1087–1105, 2021.

[52] Neil T. Dantam, Swarat Chaudhuri, and Lydia E. Kavraki. The task motion kit. *IEEE Robotics & Automation Magazine*, 25(3):61–70, 2018.

[53] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[54] Steel bank common lisp. URL http://www.sbcl.org/.

[55] James Bielman and Luís Oliveira. Common foreign function interface. URL https://cffi.common-lisp.dev/.

[56] Michael Vollmer. Newton's law of cooling revisited. *European Journal of Physics*, 30(5):1063, 2009.

[57] Clearpath Robotics, . URL https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/.

[58] Yuanchao Xu, Mehmet Esat Belviranli, Xipeng Shen, and Jeffrey Vetter. Pccs: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1282–1295, 2021.

[59] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 457–458, 2012.

[60] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 239–252. IEEE, 2020.

[61] Mohamed Hassan and Rodolfo Pellizzoni. Analysis of memory-contention in heterogeneous cots mpsocs. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[62] Wlodzimierz M Zuberek. Timed petri nets and preliminary performance evaluation. In *Proceedings of the 7th annual Symposium on Computer Architecture*, pages 88–96, 1980.

[63] Creative Commons. Creative commons legal code, 2023. URL https://creativecommons.org/licenses/by-nc-sa/3.0/legalcode.

# APPENDIX A

## RAW OUTPUT FROM ORIN POWER TESTING

Table A.1 Table of Data for time and power requirements on GPU and DLA for tasks

| Name | Time | Sys | CV | GPU | Total |
|---|---|---|---|---|---|
| Mobilenet- DLA | 4.3ms | 3680.0mW | 1121.5mW b | 2572.0mW | 7373.5mw |
| Mobilenet- GPU | 0.4ms | 2305.5mW | 858.0mW | 15936.0mW | 19099.5mw |
| Resnet152- DLA | 67.0ms | 2370.5mW | 975.0mW | 1408.0mW | 4753.5mw |
| Resnet152- GPU | 2.9ms | 3872.0mW | 726.5mW | 26658.0mW | 31256.5mw |
| Googlenet- DLA | 11.4ms | 2008.0mW | 1033.5mW | 1760.0mW | 4801.5mw |
| Googlenet- GPU | 0.9ms | 1921.5mW | 1299.5mW | 12592.5mW | 15813.5mw |

# APPENDIX B

# COPYRIGHT PERMISSIONS

## B.1 Wolfram Alpha

Figure 2.1 is licensed under CC BY-NC-SA, and adaptations or collections are permitted for non-commercial use. See section 3 of the Creative Commons Legal Code [63]. This figure has been modified by cropping the borders of the image showing path generation parameters.