RESOURCE- AND PHYSICAL-CONSTRAINT-AWARE SCHEDULING AND MOTION PLANNING, FOR CYBER-PHYSICAL SYSTEMS WITH HETEROGENEOUS PROCESSING UNITS

by

Justin McGowen

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: _____

Justin McGowen

Signed: _____

Dr. Mehmet Belviranli
Thesis Advisor

Signed: _____

Dr. Neil Dantam
Thesis Advisor

Golden, Colorado

Date _____

Signed: _____

Dr. Iris Bahar
Department Head
Department of Computer Science

ABSTRACT

Cyber Physical Systems (CPS) such as robots or self driving cars have strict requirements on both computation and physical operation to avoid failure. Heterogeneous (multi-accelerator) systems are also becoming commonplace in many CPS applications due to their ability to accelerate computational workloads, with different accelerators being optimal for different tasks.

A common task for such a CPS is to navigate a space, while running heavy computational workloads. We investigate how CPS performance can be improved both by considering physical constraints for workload scheduling decisions, and by considering computational decisions for motion planning.

The computation and physical operation of a CPS are intertwined—for example, physical obstacles can change computational latency requirements for braking safely, or the computational power draw might limit available power to motors. The motion of a CPS can also cause these requirements to vary.

These influences go both ways, and so performance is improved by simultaneously considering computation and physical operation. This presents two challenges. First, how do we find efficient schedules for CPS with heterogeneous processing units, such that the schedules are resource-bounded to meet the physical requirements? Second, how do we determine time or energy efficient motion paths when different paths vary not just in length, but also in the schedules satisfying the physical requirements?

We present a starting point towards addressing this problem: the Constrained Autonomous Workload Scheduler (CAuWS). CAuWS, with sufficient information and profiling prior to planning, can represent many hardware devices, environments, and constraints to generate schedules and support future motion planning work. CAuWS novelly considers physical constraints, computational constraints, and heterogeneous scheduling at the same time. By formalizing this problem, we also enable future motion planning work that considers scheduling during motion planning.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thank you to my advisors Mehmet Belviranli and Neil Dantam for helping me with this work, Ismet Dagli for providing profiling data, help with baselines, and determining experimental runtimes of the plans, and my committee for their time.

CHAPTER 1

INTRODUCTION

Many Cyber-Physical Systems (CPS), such as robots, autonomous vehicles, and quadcopters, must run computationally intensive workloads. To accommodate this, CPS are now being equipped with multiple specialized accelerators, sometimes on the same chip. These accelerators can be run in parallel, and often present tradeoffs in the amount of time and energy they would take to complete the same operation. When multiple tasks with dependencies must be computed, finding good assignments of tasks to accelerators becomes difficult. Simultaneously to these assignment decisions, some CPS are also making motion decisions. Given the complexity of environments, these decisions are also non-trivial.

Classically, these problems have been solved separately, but *there is benefit to be gained from considering both scheduling and planning simultaneously.* The results of a heterogeneous workload and a motion plan both depend on shared time and energy. Thus, from a Hetoregeneous Systems perspective, it is possible to create more time- or energy-efficient schedules by considering the scheduling requirements of a motion—conversely from a Motion Planning perspective, plans can be improved by considering what schedules (and their associated costs) a motion will require.

We present a novel scheduling approach that generates optimal schedules (assignments of tasks to processors, with dependencies) for CPS. These schedules adhere to resource bounds induced by the physical constraints of a CPS, which are often necessary for safety. Our approach also allows us to optimize within these resource bounds. We then describe a Motion Planning approach that could use these schedules to improve the movement time (and energy) of a motion path by considering how the available schedules change under the varying physical environments of a path.

To establish the tradeoffs that allow us to do this, consider the operation of a Herogeneous computer on a Cyber-Physical System. For many tasks run on CPS, Heterogeneous computing systems offer improved latency and energy use over a single device due to parallelism and specialization. However, to use this capability while under the physical requirements imposed by a CPS requires a way to consider both the resource use of a schedule, and the physical constraints of a CPS and environment.

CPS have two fundamental factors that limit their ability to respond to resource requirements and safety constraints. The first factors are physical—e.g., braking distance or battery charge. These requirments can vary over a CPS's operation—e.g., as obstacle distance and current battery level change. Motion planning has some control over these requirements by choosing how to move through the environment.

The other limiting factors are computational—i.e., moving and processing data to make a decision takes time and energy. Many recent CPS each embed a powerful heterogeneous System-on-a-Chip (SoC), such as NVIDIA's Xavier [1] or Tesla's FSD [2], which can improve these computational limits. Such SoCs employ a variety of processing units (PUs): often a general purpose CPU and various, specialized, domain specific accelerators (DSAs). Frequently, the time optimal PU is not the same as the energy optimal PU.

When combining the limits from these physical and computational factors, upper bounds can be placed on the resource use of CPS with time-critical components or strict constraints on energy and power. If computation takes more time, energy, or other resources than permitted, such a system might fail. For example, an aerial drone may fly at 50mph, when—considering the true computation time—it only has enough time to react if flying at 40mph. These problems can be addressed through both scheduling (by using a lower latency schedule assigning a task to GPU instead of DLA, for instance) and motion planning (by flying slower in the first place). Within such requirements, there may still be a variety of objectives to optimize (e.g., minimizing power while maintaining some latency, flying as fast as safely possible, or minimizing latency).

1.1 Scheduling

The first portion of our work addresses the challenge of *accounting for physical constraints while scheduling computational workloads in heterogeneous architectures embedding different type DSAs.*

Scheduling can make meaningful tradeoffs by assigning tasks to processors, as many CPS use DSAs for critical computations such as object detection, object tracking, or matrix operations, For example, the Xavier platform embeds a high-throughput graphical processing unit (GPU), energy efficient deep learning accelerators (DLA), and programmable vision accelerators (PVA). While the GPU can run object detection, tracking algorithms, and other vision tasks with minimal latency and high throughput, the DLA runs the neural networks (NN) used for state-of-the-art object detection (among other tasks) with half of the energy of the GPU, at the expense of a higher latency.

With these tradeoffs, scheduling while attempting to optimize objectives is non-trivial. The problem of scheduling with objectives is NP-Complete even for single CPU. This challenge exacerbated when schedules must also meet constraints, and plan on heterogeneous architectures with anumber of diverse DSAs. In some scenarios, multiple requirements (e.g., minimizing time and energy) may result in a multi-objective optimization problem [3]. We support linear tradeoffs in these cases.

Three particular limitations of existing approaches we overcome are:

- State-of-the-art computational workflow modeling techniques for heterogeneous architectures [4–8] *do not address the relationship between the physical dynamics of autonomous systems and the parallel and diverse*

capabilities of heterogeneous hardware [9, 10].

- The traditional real-time abstractions of fixed deadlines, process/task priorities, and priority inheritance in current autonomous systems not only present challenges for achieving precise timing [11], but *such abstractions are wholly insufficient to express the varying performance and energy characteristics of heterogeneous PUs* and are not capable of satisfying longer term timing requirements when run on these systems.
- Studies that map physical constraints to hardware decisions are ad-hoc, and often limited to specific constraints, optimization criteria or architectures [12, 13]. Additionally, these works represent computation as a single task, neglecting the heterogeneity necessary for optimal execution.

Overall, *the literature lacks a generalized methodology to represent the relationship between physical and computational constraints and to derive schedules optimizing desired objectives for heterogeneous CPS.*

We propose CAuWS, the Constraint-based Autonomous Workload Scheduling For Heterogeneous Architectures, which enables a generalized solution to the heterogeneous (i.e., multi-DSA) scheduling problem that takes into consideration the physical constraints of these systems with a representation language, Timed Petri nets, and mixed-integer linear programming. CAuWS, whose high-level operation is illustrated in Fig. 3.1, is able to assign operations from a wide variety of workloads to PUs on heterogeneous architectures, creating a static schedule[1] for resource-constrained and time-critical systems. These schedules are optimal with respect to the user-defined objective (including time) and profiled executions, and the schedules maintain the constraints defined in AuWL.

To accomplish this, CAuWS makes the following contributions:

- To generally bridge between diverse hardware specifications, computation scheduling constraints, and physical concerns, we propose a formal representation of that consolidates physical constraints, heterogeneous computational resources, and latency based on task assignment to PUs. We propose a timed-Petri-nets-based representation for data flow, parallelism, and resource consumption.
- By having a formal intermediate representation, we expose relevant resources and timings such that we can automatically generate constraints for both computation and physical operation. With this, we combine optimal computational schedule generation and physical system concerns—such as safety, speed, or energy trade-offs—into one optimization problem with multiple constraints and objectives. Then, we leverage existing, highly-engineered constraint solvers for mixed-integer linear programming (MILP) [14–16] to find a globally optimal schedule.

[1] For further discussion on why static scheduling is feasible (vs. dynamic scheduling) for the class of problems this work tackles, please refer to the introduction of Sec. 3.2 and Sec. 3.9.

- We use a specification language (our Autonomous Workload Language (AuWL)) as a front-end to our intermediate representation, allowing rapid specification and high level consideration of a variety of autonomous problems. AuWL specifies both data-flow and physical constraints. This application is novel compared to past languages, which either cannot represent physical constraints or are designed for hardware-software codesign (and not scheduling).
- We evaluate our approach on a simulated aerial drone. CAuWS produces a set of three schedules that allow the system to adapt to the varying physical conditions of the simulation. These schedules obey the physical constraints; in contrast, other scheduling approaches that do not consider such constraints violate physical limits for 25% of the flight and would lead to system failure.

1.2 Potential Application to Motion Planning

Safety constraints can put limits on the resource use and latency of a CPS. CAuWS can create schedules to adapt schedules to the physical environment, but motion planning can change what physical environments the CPS goes through in the first place. By incorporating CAuWS into motion planning, it is possible to find more efficient paths compared to a planner that ignores scheduling considerations.

Motion planning, when applied to robots or other mobile CPS, finds a path through an environment. CAuWS provides a solution to finding the best schedule for different parts of this environment, allowing us to adapt schedules over the course of a fixed path.

However, by having CAuWS as a complete system that allows us to query possible schedules, CAuWS can be incorporated as a subsystem in motion planning. Some parts of the environment may not have any valid schedules—while a standalone Motion Planner may not know this, CAuWS can avoid these areas.

Furthermore, even if there are valid schedules in all areas, there can still be tradeoffs. An area that is denser in or closer to obstacles may require low latency to stop in time. Alternatively, a CPS could just plan to move slower through this area, or even path around it entirely. To effectively compare these different paths, there must be a way to consider what paths have valid schedules, how computation latency affects maximum speed, and other such considerations. CAuWS can provide a means to make these considerations. By incorporating CAuWS's schedules into Motion Planning, it is possible to find paths that use less time or energy.

While past Motion Planning approaches are general (and can incorporate CAuWS), no work has proposed incorporating such schedules when evaluating their approaches. There are works that consider computational and physical resource use as a criteria for Motion Planning, but these still do not consider schedules. However, experimental demonstration of our proposed technique remains future work

CHAPTER 2

RELATED WORK

Scheduling for a variety of workloads on heterogeneous systems has been investigated over the last decade. A vast majority of key studies [17–21] are dynamic and typically use task-based heuristics. Static scheduling for heterogeneous processors [22–25] is also common, with an emphasis on polyhedral code generation [26, 27] and genetic algorithms [28]. Our proposed approach significantly differs from these studies by its ability to map the physical dynamics of the system to the performance characteristics of heterogeneous computing.

A limited number of studies have structurally approached timing in CPS computation by building models relating physical constraints and computational elements. For example, Krishnan et al. [12] create a computational model to map processing power to the weight of a CPS, and Wan et al. [29] focus on the computational resilience of navigation systems. However, approaches proposed by these studies are restricted to a specific physical constraint and do not address general, domain-independent criteria. Most recently, Hadid et al. [13] introduced a design-space methodology to explore the effects of a wide range of physical factors on compute scheduling; however this work also does not address generalized techniques for CPS scheduling problems.

Petri nets are a form of directed graph that offers a convenient representation of dependencies, parallelism, and resources. A limited number of studies considered Petri nets for CPU-based, formal scheduling representations for real-time [30–32] and hybrid [33, 34] systems. Relevant to this paper are works establishing Timed Petri nets [35], works that use these to represent heterogeneous embedded systems [36], and works using Petri nets for scheduling [37, 38]. The works involving embedded systems used Petri nets for verification purposes [36, 39]. While Petri nets have been used for scheduling [38] on single cores, they have not been used for *scheduling* of *heterogeneous* systems yet, to our knowledge. Our proposed framework uniquely extends timed Petri nets to specify complex scheduling constraints introduced by heterogeneous PUs.

Constraint solving is a widely-used technique in a number of fields including automated planning [40], robotics [41], and program verification and synthesis [42, 43]. Critically, modern, highly-engineered constraint solvers offer a variety of techniques to efficiently address computationally hard problems [14, 15]. Previous scheduling techniques have also used constraint solving in an ad-hoc manner. Additionally, many works with Petri nets have also applied constraint solving.

Motion planning at its most general is a means of finding a path through a space, where the space is subdivided into an obstacle region and a free region. This is practically applied to robots which need to move

from a starting position to a goal position. This can be further complicated by attempting to find better motions—in the context of robots, typically in path length or time. These approaches have been extensively researched—for an overview, see [44–46].

We specifically propose an RRT planner [47, 48], based on the control-space RRT planner in OMPL [49]. This method randomly expands a tree of possible motions. To find better paths, this could further be extended to an RRT* planner [50] by rewiring the tree as better paths are discovered. [51] provides a further overview of RRT-based methods.

However, no motion planning works have yet evaluated how *scheduling decisions* can affect the resulting motions. There are works performing the reverse, by proposing scheduling systems that can adhere to hard deadlines and resource limits for autonomous vehicles [52, 53] or more general systems [54]. As safety requires hard limits, these could be used with a motion planner to guarantee resource usage, but further improvements can be had by knowing the schedules at the time of planning.

Numerous works still do consider resources in motion planning. Some works consider how to maintain communication over the planned path(s) [55, 56] or distributing planning across multiple robots [57]. These both must make considerations beyond just finding the shortest path. There are also works investigating Energy-Aware Motion Planning [58–60], but these consider solely the energy of motion, and not of computing energy. The closest work we know of is [61], an exception that consider computation energy and makes a compelling case for it, but this work considers energy in terms of when to terminate repeated iterations of the planning algorithm itself, instead of scheduling other operations like CAuWS.

Lastly, there are works investigating the design of chips for motion planning [62, 63]. These works do consider both parallelism and motion planning, but still do not consider physical constraints or resource requirements. More broadly, the areas of synthesis and codesign for Cyber-Physical Systems also considers physical concerns when designing systems [64, 65], but even when heterogeneous these do not typically consider scheduling either. These works can consider a range of problems, and make decisions based on physical concerns, but do not have the adaptability and adherence to constraints that CAuWS can provide for Scheduling and (potentially) Motion Planning.

CHAPTER 3

CAUWS: PROPOSED METHOD

3.1 Problem Description

All robots have some limitations from being a physical object in an environment. Whether because of compute latency, energy requirements, safety considerations for operation, or simply the inability to phase through objects—some motions, positions, or states are not permitted. Likewise, for a given computation workload and computing device, the available performances of schedules (in terms of latency and energy use) are limited. For heterogeneous systems running a number of computation tasks, the available tradeoffs between computational latency and energy are largely determined by how tasks can be assigned to processors. Generally, the energy and latency of these choices are inversely related. Notably, these limitations can both impact each other — and this link can go both ways depending on what we have control over. When choosing schedules of processor assignments, the physical limits place restrictions on schedules we can choose. For example, with a choice between a GPU and a DLA or vision accelerator, we can assign tasks to the GPU for lower latency (if a provided path starts moving faster, lower latency may be required to allow the robot to brake in time). The other device can be used when less energy is desired (a hot environment may require efficient schedules to avoid overheating) Conversely, the choice of the physical environment the robot is in (or rather, the optimal motion plan through parts of this environment) can be impacted by the set of possible schedules. Knowing the latency and energy tradeoffs, plans can go faster in some areas, or choose to avoid areas that require lower latency.

We describe a method for performing both of these considerations. While these methods are general and can apply to many environments, physical constraints, workloads, and robots, it helps to define a concrete example of these parameters here. We also provide the methods and values of experimentally determined constants.

These are the input to our method, the limitations of the CPS—the robot, the constraints the environment places on the robot, and the workload the robot must compute. Once these are established, we first create a system, the Constrained Autonomous Workload Scheduler (CAuWS) that can create optimal computational schedules for any valid point in a robot's physical environment. This in turn enables motion planning that considers these schedules when calculating the cost of paths.

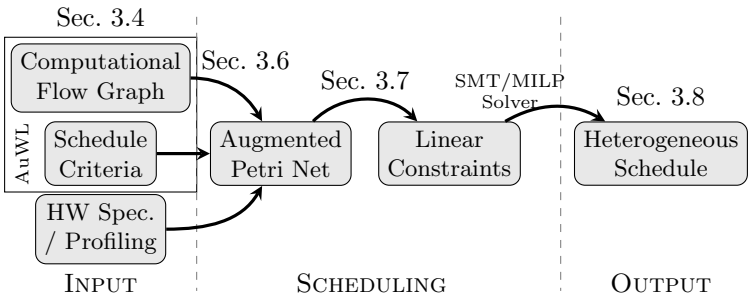

Figure 3.1 Overview of CAuWS.

3.2 Overview

In this section, we explain CAuWS, our proposed, novel scheduling methodology for heterogeneous CPS. The input to CAuWS is a specification that includes (1) *the control flow graph* (CFG) represented using operations in the AuWL file, (2) the necessary *performance criteria* (e.g., a constraint on time, energy or power), and (3) the *profiling data* for estimated running times and energy consumption of tasks on available PUs. The output of CAuWS is a heterogeneous schedule that includes (1) *the set of tasks*, (2) *the ordering of tasks*, and (3) *the mapping of tasks to PUs*.

CAuWS first uses the system specification (CFG, performance criteria, and profiling data) to construct a Petri net as an intermediate representation of the CPS. Importantly, the Petri net captures the parallelism and dependencies of tasks, data, and resources (PUs) in the system. Valid firings of this Petri net correspond to valid schedules. Then, our method uses the Petri net to generate a set of constraints and objectives corresponding to valid and optimal (with respect to the objective) schedules. We find a solution to these constraints using a state-of-the-art solver (specifically, Z3 [14, 15]) to obtain a heterogeneous schedule that satisfies the physical requirements. Z3 guarantees optimality given an objective, selecting from the wider set of Pareto-optimal schedules.

3.3 Assumptions for the Validity of Static Scheduling

Many CPS workloads permit *two simplifying assumptions* that CAuWS makes based on prior knowledge of the workload.

First, many CPS workloads operate at discrete time steps, e.g., running a set of tasks per image frames captured at a fixed frequency. These time steps introduce natural synchronization points after a set of tasks. In contrast, some dynamic scheduling approaches maintain a queue of tasks to execute. By knowing all tasks up to the synchronization point, CAuWS finds optimal schedules for this set of tasks.

Second, for many CPS, this workload stays the same between time steps, which permits static scheduling. Many tasks CPS, such as the various computer vision networks, have a fixed size and no conditional execution. Therefore, these tasks can be profiled prior to runtime. Furthermore, the physical construction (i.e., sensors and actuators) of a CPS is static, so the inputs to these tasks often do not change.

Notably, these two assumptions enable CAuWS to produce a set of static schedules that can handle *dynamic conditions*. As an example, some CPS often have a limited number of modes—a drone may have one mode to locate and object of interest and another mode to monitor that object. Other times a CPS physical environment can change (with that change measured by sensors), and a CPS may want to adjust scheduling based physical parameters such as temperature or distance. CAuWS can generate sets of schedules for each of these cases, and then dynamically switch schedules at runtime. That is, statically creating a policy that allows the CPS to adapt its scheduling dynamically. We detail this process in Sec. 3.9.

3.4 Specification of Input

Our proposed system introduces the Autonomous Workload Language (AuWL) representation. AuWL describes the data flow of tasks (i.e., the CFG) and the necessary schedule criteria (e.g., minimizing computation time, meeting an energy budget). This rich specification goes beyond traditional scheduling abstractions and is necessary to correctly satisfy physical constraints and heterogeneous flows. Furthermore, while past specification languages exist, the ability to specify constraints and flow in a single file allows rapid configurability and a high level abstraction of the underlying constraints that is independent of underlying hardware.

```
model AuWL_example {
    constraint (= total_power 30)
    constraint (= velocity 5)
    constraint (= motor_power (* velocity velocity))
    constraint (< ENERGY 50)
    constraint (< POWER (- total_power motor_power))
    objective  (- TIME)
    data camera, lidar
    data obj_bounding_boxes, localized_position, route
    op object_detection {in=camera;out=obj_bounding_boxes}
    op localization {in=lidar;out=localized_position}
    op route_planning {in=obj_bounding_boxes, localized_position; out=route}
}
```

Figure 3.2 An example AuWL program, defining the primary operations, their dependencies, CPS constraints and the objective for a simplified model of an autonomous vehicle.

Fig. 3.2 contains an example AuWL representation for a simplified autonomous scenario, which must minimize execution time under several constraints. The tasks `object_detection` and `localization` process camera inputs and lidar inputs and then output the data `object_bounding_boxes` and `localized_position` used by task `route_planning`. The `constraint` keyword enables users to symbolically represent physical properties. The `motor_power` is the square of `velocity`. POWER is the remaining power budget for computation, which is the difference between `total_power` and `motor_power`, and ENERGY (the integral of POWER over TIME) is the remaining energy budget for computation. The objective in this example is to minimize TIME within the computation limits imposed by `motor_power`. Higher `motor_power` use will leave less power for computation, resulting in lower-energy PUs being preferred during scheduling.

Our scheduling method also incorporates information about the hardware—i.e., the available PUs—and timing information about each task and any PU on which it may run. The timing information enables *CAuWS* to ensure that specified constraints and objectives are achieved. For the case studies, we determine these times empirically by profiling the tasks in the workload separately. Developing precise performance models for autonomous systems is outside the scope of this study.

3.5 Petri Net Background

Before discussing the detailed generation of Petri nets in CAuWS, we provide a review of the key details of Petri nets and the extensions we use to formally represent heterogeneous scheduling[2].

A Petri net is a directed, bipartite graph.

Definition 1 *A Petri net is the tuple* $\mathcal{N} = (P, T, E)$*, where,*

- P *is the finite set of* place *nodes,*
- T *is the finite set of* transition *nodes,*
- $E \subseteq (P \times T \cup T \times P)$ *are the* edges *between places and transitions.*

Each place P may contain a number of *tokens*. We call the number of tokens contained in all places a configuration or *marking* of the Petri net. When a particular transition *fires*, it changes the marking by decrementing the tokens at incoming places and incrementing tokens at outgoing places.

Petri nets are often a convenient model to represent shared resources in parallel systems. Places represent a particular resource, and the token count at a place represents how much of that resource is available. In our scheduling application, we use places to represent the availability of a PU. Transitions specify the possible changes in resources. The incoming places to a transition represent resources that are acquired or consumed, and the outgoing places represent resources that are released or produced. The Petri net captures

[2]For thourough coverage of Petri nets, we refer the reader to texts such as [66].

the parallelism of many systems by allowing transitions to fire in any order, as long as their incoming places have positive token counts.

Our scheduling approach uses a mixed-integer extension to classic Petri nets. While classic Petri nets are discrete, many real systems have continuous resources, such as energy. Thus, we use Petri nets where some places may have a real-valued number of tokens.

Each edge of the mixed-integer Petri net has a weight indicating the number of tokens moved. When a transition fires, it removes from its input places and adds to its output places a number of tokens equal to the corresponding edges' weights. Token counts must be still be non-negative, so transitions may only fire when places contains token counts of at least the corresponding weight.

Lastly, we apply Timed Petri nets, which incorporate a delay time for each transition. Before a transition may fire, all its input places must contain the necessary number of tokens for this delay time. We use the delays to represent computation time.

3.6 Petri Nets as an Intermediate Representation

We use the specification given in Sec. 3.4 to construct a Petri net intermediate representation (see Fig. 3.4). The Petri net captures the structure of control flow choices and resource use, facilitating the subsequent generation of constraint by exposing the relevant resources.

First, we construct the Petri net places representing shared resources and available data. We construct one place for each PU and each data element. A token in a PU place indicates that the PU is available; when a task is running on a PU, the token must be removed from the corresponding place. A token in a data element place indicates that this data element is available. Some data element places will correspond to CPS inputs (e.g., raw sensor readings) or CPS outputs (e.g., actuator commands).

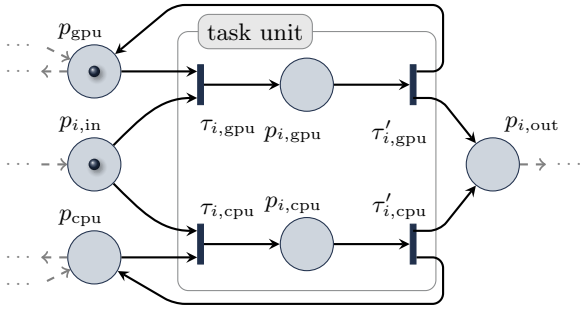

Figure 3.3 Petri net "unit" for one task. Each unit has multiple "paths", one for each PU. The unit operates by (1) firing $\tau_{i,xpu}$ to consume tokens for its input and a selected PU and setting a token on the place to remember the selected PU, and (2) firing $\tau'_{i,xpu}$ to set a token in the output place and restore the PU availability token. In this example, the GPU is available.

Then, we construct additional places, edges, and transitions according CFG defined in the AuWL specification. This construction consists of multiple units in the form of Fig. 3.3, resulting in an overall Petri net such as Fig. 3.4. For each PU to which a task i may be assigned, we create (1) a place ($p_{i,\text{xpu}}$) indicating task i is running on PU xpu, (2) a transition $\tau_{i,\text{xpu}}$ indicating the task starting on the PU, and $\tau'_{i,\text{xpu}}$ indicating the task finishing on the PU. We create edges that the describe the availability of the PUs, the input, and the output. We create these units for every task in the AuWL specification.

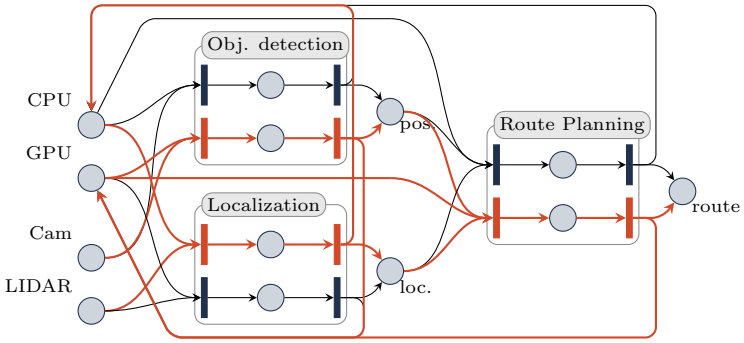

Figure 3.4 Proposed Petri net construction. An example Petri net for a minimal CFG with multiple units and layers. Highlighted paths indicate a valid firing sequence. The highlighted firing show the GPU running object detection while the CPU simultaneously runs localization. Then, the GPU runs route planning.

Finally, we augment the Petri net with the schedule criteria (constraints and objectives) from the AuWL file and the timing information from the hardware specification. Some schedule criteria may correspond to time constraints (e.g., ensuring adequate reaction/computation time based on a minimum stopping distance), which we add to the Petri net as a constraint on final time, $t^{\langle \text{end} \rangle} \leq \text{const}$.

Other schedule criteria may indicate shared resources, such as available power or energy. This capability is an advantage over typical CFG-only system specifications. We model a shared resource by creating additional place p_{res}. Then, we create edges for transitions that use this resource. For example, running a task on a PU requires a certain amount of power. The transition to start this task $\tau_{i,\text{xpu}}$ has incoming edge $(p_{\text{power}}, \tau_{i,\text{xpu}})$ and the transition to finish the task has outgoing edge $(\tau_{i,\text{xpu}}, p_{\text{power}})$. Both these edges have a weight equal to the power required to run the task on the PU.

Finally, we specify the valid initial and final markings of the Petri net. At the initial timestep, any CPS inputs are available, and at the final step, any CPS outputs must be available.

$$\forall p_{\text{in}},\ \left(p_{\text{in}}{}^{\langle 0 \rangle} = 1\right) \quad \text{and} \quad \forall p_{\text{out}},\ \left(p_{\text{out}}{}^{\langle \text{end} \rangle} = 1\right) \tag{3.1}$$

At the initial timestep, any resource places contain an initial value of $p_{\text{res}} = \text{const}$.

3.7 Petri Net Constraint Generation

We use the Petri net to construct a set of constraints for valid schedules. A solution to the constraints corresponds to a sequence of Petri net firings and a heterogeneous schedule.

3.7.1 Marking Constraints

First, we construct constraints for subsequent markings (token counts) of the Petri net. The key constraint is that a firing transition removes tokens from its input places and adds tokens to its output places, which we define in terms if inflow and outflow at each place:

$$p^{\langle k+1 \rangle} = p^{\langle k \rangle} \quad + \overbrace{\sum_{j=1}^{|T|} {\tau_j}^{\langle k \rangle} W(p, \tau_j)}^{\text{inflow}} \quad - \overbrace{\sum_{j=1}^{|T|} {\tau_j}^{\langle k \rangle} W(\tau_j, p)}^{\text{outflow}} , \tag{3.2}$$

where $p^{\langle k \rangle}$ is the token count of place p at step k, ${\tau_j}^{\langle k \rangle}$ is true if transition τ fires at step k, and $W(x, y)$ is the weight of the edge from node x to node y.

We additionally constrain each place to have a non-negative token count, $p^{\langle k \rangle} \geq 0$, and we add constraints to ensure valid initial and final markings (as specified in Sec. 3.6).

3.7.2 Timing Constraints

Next, we construct constraints for the timing information. A transition must be *enabled* before it can fire. Transitions can only be enabled if all input places contain sufficient tokens. After a transition has been enabled for its delay time, it will fire and then be *disabled*.

We create additional variables in the constraint formula to account for timing. Real variable $t^{\langle k \rangle}$ represents the (continuous) time at (discrete) step k. Boolean variable ${e_j}^{\langle k \rangle}$ indicates that transition j is enabled at step k. Real variable ${u_j}^{\langle k \rangle}$ indicates the time at which transition j was enabled.

The timing constraints ensure the validity of transitions being enabled, firing, and disabled according to token counts, edges, and time delays.

3.8 Schedule Generation from Constraints

Finally, we solve the constraints to obtain a schedule. We use a solver for Satisfiability Modulo Theories (SMT)—specifically, Z3 [14, 15]—to solve the constraints from Sec. 3.7. The solution to the constraints corresponds to a sequence of Petri net firings and a valid heterogeneous schedule. The schedule is encoded in the transition firings ${\tau_{i,\text{xpu}}}^{\langle k \rangle}$ and times $t^{\langle k \rangle}$. When the variable ${\tau_{i,\text{xpu}}}^{\langle k \rangle}$ is true, the schedule will assign task i to PU xpu at time step k, occurring at real time $t^{\langle k \rangle}$. We collect all such true firing ${\tau_{i,\text{xpu}}}^{\langle k \rangle}$ variables and all times $t^{\langle k \rangle}$ to determine the full heterogeneous schedule.

3.9 Handling Dynamic Conditions via Static Scheduling

Our proposed methodology in Sec. 3.2 produces optimal schedules only if *computation time* and *physical factors* (e.g., velocity, battery) can be considered statically, before execution. CAuWS produces schedules from an MILP which is not feasible to solve dynamically during the operation of a CPS. Table 3.1 shows how Z3 solver runtimes are affected by schedule complexity. This static assumption may be considered limiting—however, we explain in this section how we work around this to handle dynamically changing computational and physical aspects an adaptive set of static schedules.

3.9.1 Computation Time

Computation time in most CPS scenarios depends on the following factors (and assumptions) which *can* be known beforehand:

- Input size: Typically, the input data is provided by fixed resolution devices, such as cameras or lidar. Moreover, many neural networks that CPS rely on operate on a fixed image or video frame sizes.
- CFG topology: While some CPS scenarios have branching computational flow graphs to handle conditional computation, these branches can often be split into separate static CFGs. The specific branch followed in the CFG often depends solely on either the physical mode of the system (e.g., running accurate NN or faster blob-based detection based on whether the drone is in discovery or tracking mode) or on pre-determined time intervals (e.g., running a complex object detection every 10th frames and simpler tracking in-between).
- Scene complexity: For some CPS tasks, such as route- and motion-planning in autonomous driving and robotics, the number of objects in a scene can increase how much computation is needed. On the other hand, some tasks, like the vision networks used in the case study Sec. 4.5, require the same amount of computation regardless of the current input.

Table 3.1 How the runtime of CAuWS scales as the size and complexity of problems increases. While these times may appear long, CAuWS supports precomputation of multiple schedules that can adapt to dynamic conditions or operation modes, avoiding any overhead at runtime. Further engineering of constraints, e.g., similar to Rintanen et al. [67], remains an area of future work.

	Number of Accelerators					
	2			4		
Parallel Paths:	1	2	4	1	2	4
4 total tasks	0.459	2.187	17.952	2.079	1.764	6.888
8 total tasks	4.643	6.548	52.112	62.707	5.908	15.775

When *computation time* can be predicted statically, statically creating schedules becomes possible, and with significant benefit: pre-computing schedules can take as much time as necessary, allowing CAuWS to select only (valid) Pareto-optimal schedules.

3.9.2 Physical Factors

Physical factors likewise are a challenge when creating static schedules. Cyber-physical systems operate in environments with *changing* physical conditions (e.g. current distance to obstacles, ambient temperature).

CAuWS uniquely handles dynamically changing physical factors by pre-computing multiple schedules over a range of physical values (e.g., the system must use a faster schedule when closer to obstacles, and a more energy efficient schedule otherwise). We identify the borders between different schedules in terms of the physical factors—i.e., the switchover points where changing a physical parameter causes a different schedule to best satisfy the constraints.

While searching for these schedules and their borders in the physical parameter space, it is important to limit the number of solver invocations. As previously shown in Table 3.1, finding even a single schedule is costly. As such, the feasibility of a naïve grid-based discretization of the physical values is limited. An overly-coarse grid means the boundaries will be inaccurate, and the schedules generated will result in suboptimal resource usage, or, at worst, physical failure. However, the complexity of a grid discretization scales poorly in both dimensionality and resolution, and so reaching sufficient resolution takes prohibitively long. Instead, we use a recursive, binary partitioning of the n-dimensional hyperspace of physical parameters.

3.9.3 Linear Spaces for Varying Schedule Inputs

To formalize the problem, we consider the schedule output by CAuWS as a function of input parameters, which are the physical quantities in the system that vary, such as velocity, obstacle distance, and temperature. These physical parameters can be considered as an n-dimensional hyperspace. For visualization, we can plot these dimensions (for a small value of n) as the axes of a graph. Fig. 3.5 (pg. 17) depicts a slice of one such hyperspace ($n = 3$) and how it is partitioned into different schedules. Furthermore, finding regions for valid schedules enables handling of some nonlinear constraints, which cannot be directly translated to a MILP. For example, the velocity constraints in Equation 4.2.3 are nonlinear (v^2). However, if we take velocity as a given for a single check from the MILP, both v and v^2 become constant for the duration of the MILP. With this, we can create optimal schedules for all values of v by finding the boundaries where different v's change the resulting schedule.

A key property of these schedule spaces is that, under certain assumptions, we only need to check that the vertices of a schedule region are the same to guarantee that every schedule inside the region is the same.

Theorem 3.1 *If all points on the border of a convex region R in a hyperspace share the same valid schedule, and as each input parameter (taken separately) varies all physical constraints are either more or less satisfied (i.e. the constraints are all monotonic w.r.t. some parameterization of inputs), then the entirety of R must*

share the same schedule.

Proof: Given these input parameters as a basis, for any point P in R we can define a path L through P that starts and ends on the border of R along these bases, such that all constraints are more strongly satisfied along L. If P did not share a schedule, then the schedule must change twice along L —but this would require some constraint to be violated, which is impossible as all constraints are monotonically more satisfied.

This holds for the experiments as all constraints (Sec. 4.2) are monotonic with respect to obstacle distance, velocity, and temperature. Stopping Distance strictly increases with velocity, power usage strictly increases with velocity, and maximum heat generation strictly decreases with ambient temperature.

3.9.4 Algorithm to Find Schedule Boundaries

To avoid the number of CAuWS invocations a grid-based discretization of the parameter space would cause, we propose a binary space partitioning algorithm to find schedule boundaries. The MILP formulation in CAuWS limits optimal scheduling queries to a single parameter point. Thus, we identify convex regions for a valid schedule by finding sets of vertices that share the same schedule.

```
Divide each dimension in half to create 2^n hypercubes
Sample all corners of the hypercubes
By the theorem, if the corners of a hypercube share
    the same schedule, the whole hypercube does,
    and the schedule can be returned
If the corners of a hypercube are not the same,
    recursively subdivide and repeat
Terminate as a base case when hypercube
    is smaller than some desired tolerance
At runtime, check which hypercube the
    desired point is in and get the corresponding schedule
```

Listing 3.1: Binary-space partitioning

We identify the scheduling regions using the binary space partitioning approach given in Listing 3.1. Based on Theorem 3.1, this algorithm checks the vertices of a hypercube. If all vertices have the same schedule, then that schedule is valid throughout the hypercube. Otherwise, we recursively subdivide the hypercube down to a given minimum resolution. The result is a set of hypercubes of varying sizes, each corresponding to a schedule, covering the space.

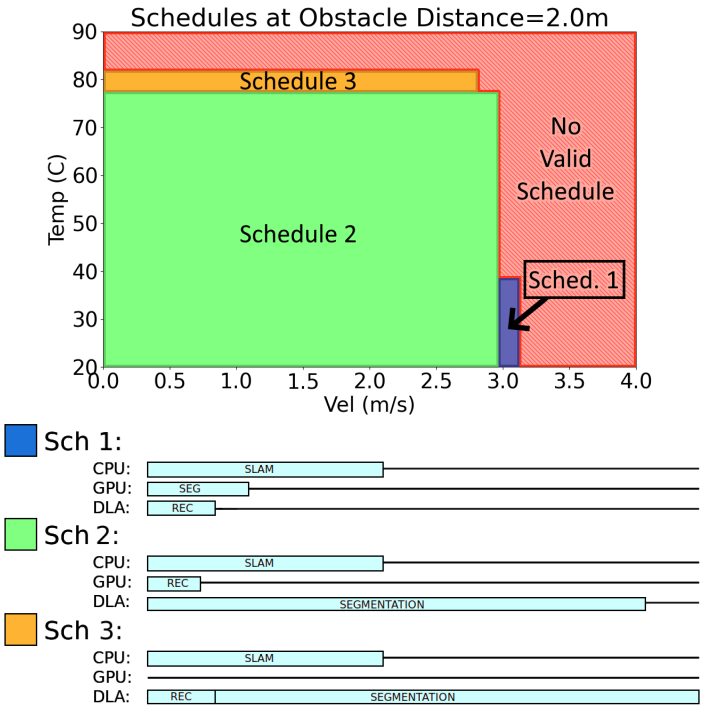

Figure 3.5 The set of schedules generated for the case study. CAuWS can precompute multiple schedules for varying physical parameters, creating a policy such as "if speed is less than 3m/s and ambient temperature less the 76, run NN 1 on the GPU and NN 2 on the DLA simultaneously". The additional constraint of no idle time was enforced. This particular set of schedules is a slice that applies when the drone's distance to obstacles is 2m. Schedule 1 is fast but uses more energy, while Schedule 3 is slow but more energy efficient, due to the choice of processors.

CHAPTER 4
CAUWS: EXPERIMENTS

4.1 Robot Setup

For the experiments, we evaluate our approach on a simulated quadcopter drone. The drone has a camera and an Nvidia Xavier AGX or NX SoC (depending on the experiment), both notably containing a multi-core ARM CPU, a Volta based GPU, and a Deep Learning Accelerator (DLA).

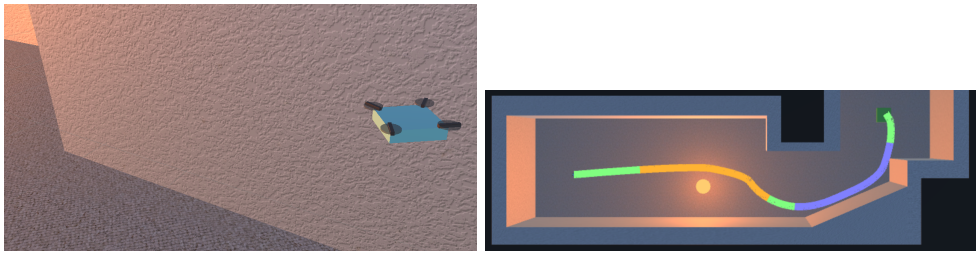

Figure 4.1 A screenshot of the simulation. This includes quadcopter dynamics for a simulated 3DR-Iris quadcopter drone, positional heat sources (the orange dot), and obstacles. The highlighted path is colored based on what schedule CAuWS chooses, with different choices when close to the wall or heat source. These schedules correspond to those in Fig. 3.5.

The drone must run the following tasks to complete its objective: (1) An image identification network to identify humans in need of rescue in order to contact help, (2) an image segmentation network to parse the environment into walls, doorways, and fires, and (3) Simultaneous Localization and Mapping (SLAM) to know its position in the environment (which can also correlate with the segmentation). These tasks were profiled offline on the AGX in terms of computation time and energy. This creates a workload with dependencies that CAuWS must then schedule, in turn supporting MACAuWS to consider the impact running these tasks has on its motion planning.

Once the drone has the output from these tasks, it can plan its future actions on a solo CPU. We neglect this task as the time and energy it contributes is negligible, especially when run in parallel to the more intensive task. As CAuWS pre-computes schedules before operation, the cost of selecting the appropriate schedule is also negligible.

4.2 Physical Constraints and Approximations

To expose the physical limitations to computational scheduling (and, in turn, motion planning), we express the limitations as a set of inequalities that must be satisfied. These inequalities allow the environmental, robotic, and computational limitations to be jointly considered, as many of the equations

depend on values from more than one of the limitations. For example, the braking distance safety constraint (Equation 4.2.3) relates robot velocity, obstacle distance, and compute latency—the overheating constraint (Equation 4.2.1) ambient temperature and compute energy.

Constraints can be more general than a specific setup, and many more such constraints can apply to other problems. However, the setup we use for experiments is largely the same for both the schedule generation and future motion planning we consider, and both share constraints even if they do not share specific parameter values. As such, to motivate the other sections, we define the specific environment and constraints here—we also provide values for the parameters that come from a robot specification or were experimentally determined.

The environment we consider induces a set of 3 constraints for CAuWS. Our novel DSL and Petri-net based representation makes it trivial to support additional constraints for different scenarios. The constraints are:

1. *Heat:* The system cannot use too much computation power, otherwise it will overheat in the hot environment
2. *Power:* The system must schedule within the power constraints caused by variations in speed and motor power.
3. *Stopping distance:* The system must maintain a safe latency to react and stop given velocity and obstacle distance.

Additionally, when these constraints are satisfied and there is some freedom, we want to minimize latency and energy usage. The specific *linear* tradeoff between energy and latency is defined in the AuWL file (Fig. 4.2). While general multi-objective optimization can consider nonlinear tradeoffs, MILP solvers can find the global optimum for linear objectives such as this.

The constraints are expressed to CAuWS through the AuWL file (Fig. 4.2). The system has a variety of ways to respond to the constraints. The scheduling targets include a CPU, GPU, and DLA to choose between. In general, for neural network based computer vision tasks, the DLA will be more energy efficient but slower than the GPU (therefore producing less heat). Parallelism can reduce latency at the cost of maximum power. *These different responses create a set of three schedules that CAuWS chooses from over the course of the simulation* (Fig. 4.1).

Once the constraints and optimality objective are expressed in the AuWL file, CAuWS successfully produces the optimal schedule that adheres to the constraints. The latency is defined as the time it takes to run one "iteration" of the schedule—that is, to run each task once.

The set of formal constraints these physical limitations create follows.

4.2.1 Heat

As heat is a constraining factor in computation, the fires in the simulation provide a rapidly varying temperature which CAuWS must adapt to. If ambient temperatures are too hot, it must schedule tasks on the DLA instead of GPU to reduce heat produced. We place a constraint on steady-state heat flow, where the total heat flow from cooling (over the course of the schedule) must be greater than the heat generated by the schedule. Cooling rate depends on ambient temperature T_{amb}. With T_{max} as maximum operating temperature, C total specific heat, E_{op} the energy of each operation, and an experimentally determined cooling rate k, the final constraint is:

$$C \cdot \left(\sum E_{op} - k(T_{max} - T_{amb}) \cdot t^{\langle \text{end} \rangle} \right) \leq \Delta T_{sys} \leq 0 \tag{4.1}$$

This steady state heat flow is required to create a tractable linear constraint instead of the exponential constraint modelling the current temperature would cause. To derive this, we start with T as temperature and total specific heat C. The equation for heat flow is (≤ 0 for steady state):

$$\frac{dT_{sys}}{dt} = C \cdot (P_{in} - P_{out}) \leq 0 \tag{4.2}$$

P_{in} is computation power, and P_{out} is proportional to the difference in temperature.

$$C \cdot \left(P_{comp} - k(T_{max} - T_{amb}) \right) \leq \frac{dT_{sys}}{dt} \leq 0 \tag{4.3}$$

The maximum operating temperature is used for the steady state—if the flow at this point is 0, it can never exceed the maximum. While this does overestimate the flow out, it is still conservative as any lower flow will eventually reach equilibrium.

To use this constraint in CAuWS, we integrate by time. This discretizes P_{comp} to the energy of all operations per iteration, and adds the term $t^{\langle \text{end} \rangle}$ (total latency) to P_{out}. Heat changes on timescales much longer than $t^{\langle \text{end} \rangle}$, so this discretization introduces negligible error.

For the scheduling experiments, T_{max} is set to be AGX's temperature reaching the rated $85C°$. The die temperature and power were collected by reading corresponding I2C addresses on the Xavier AGX.

To determine the other constants, an intensive program is run and an exponential function fit to the resulting temperature curve (see Sec. 4.3 for more details).

4.2.2 Actuation Power

The power output of the battery at any given moment is shared between the rotors' actuation and the computation, placing a constraint between computation power and acceleration and velocity.

$$P_{battery} > P_{total} = P_{act} + P_{comp} \tag{4.4}$$

While P_{comp} is determined from schedule choice, P_{act} can further be broken down into terms representing the required power to:

- stay aloft
- maintain a velocity against drag
- maintain thrust to accelerate

$$P_{battery} > P_{accel} + P_{vel} + P_{aloft} + P_{comp} \tag{4.5}$$

$P_{aloft} = 700.7W$ was determined experimentally by hovering in simulation.

For P_{vel}, we approximate air resistance as $k_{air} \cdot v^2$ and the output thrust of a drone rotor as $k_{thrust} \cdot P_{accel}$. While hovering, the thrust must equal the weight. This gives an estimate of $k_{thrust} = 0.0210N/W$. Balancing these forces:

$$P_{vel} = \sqrt{k_{air}/k_{thrust}} \cdot \sqrt{v}\ , \tag{4.6}$$

The value of $\sqrt{k_{air}/k_{thrust}} = 7.6$ was experimentally determined by flying at a fixed velocity.

Likewise, if we are currently in an accelerating maneuver, we add an additional term:

$$P_{accel} = k_{thrust} \cdot mass \cdot a \tag{4.7}$$

4.2.3 Stopping Distance

As the compute latency increases, the stopping distance increases due to the additional "reaction" time of the drone. A given velocity thus places an upper bound on latency to react in time and avoid unexpected collisions. In these situations, some tasks can be schedule on GPU to reduce latency at the cost of energy. The basic constraint on the equation of motion is:

$$D_{obst} > D_{stop} = v \cdot t^{\langle \text{end} \rangle} + \frac{1}{2} a_{max} \cdot t_{stop}^2 \tag{4.8}$$

We can determine a maximum acceleration assuming the drone directs all power to its rotors. Using this value $a_{max} = k_{thrust} \cdot P_{battery} = 37.7622m/s^2$ we rewrite the stopping time as $\frac{v}{a_{max}}$, resulting in the equation of motion:

$$D_{obst} > D_{stop} = v \cdot t^{\langle \text{end} \rangle} + \frac{1}{2a_{max}} \cdot v^2 \tag{4.9}$$

D_{obst} varies over the simulation due to the motion of the drone.

4.3 Heat Approximations

As setting a building or robot on fire is impractical, we must make approximations to model heat for both the simulated environment and simulated robot. This second is also important for linearizing the heat constraint Equation 4.2.1

For the first, we assume a base ambient temperature. Fires then apply additional ambient temperature, with a linear falloff as distance increases. The position of a fire is fixed in simulation.While not particularly accurate to real fires, the important part to demonstrate to our approach is the variation in heat they provide, and not the exact gradient. In a real environment, heat could be estimated using an IR camera.

The second, modelling heat on the robot (specifically the SoC), requires some experimentation. We already know the heat produced by operations (as it should be 1-1 with energy consumption), but determining how this impacts temperature requires determining the specific heat C of the SoC. We also need a cooling rate of the SoC to model the heat dissipation to the ambient air. Conservatively, we neglect additional heat dissipation from convection (which could only improve dissipation aP_{out}) by experimentally finding the rate k in a still room.

To determine both C and k, we ran an intensive workload on the AGX to raise its temperature. After recording the ambient temperature, we simultaneously tracked energy consumption of the SoC, allowing us to determine how much energy is required to raise the device a degree. This resulted in an experimental average of $C = 59.3J/C^{\circ}$.

To then determine k, we fit an exponential curve to the data between time and temperature (the simplest equation modelling cooling is an exponential integrating Equation 4.2.1). From this fit, we can pull the rate $k = 0.331W/C^{\circ}$. Our experiments include two simulation case studies and two sets of smaller tests.

The following two case studies involve a *simulated* Iris 3DR drone that must run various computer vision and planning workloads on an off-the-shelf SoC. These demonstrate the range of constraints CAuWS can schedule for, and how they can improve performance.

The first environment is a search-and-rescue task where we must also consider heat limitations due to fire, while balancing energy and latency. The second is an adversary pursuit, with different priorities before and after adversary detection, while obeying power limits. These environments share the same drone and run similar vision networks, but are otherwise sepearate.

While we consider drones in these case studies to demonstrate a more compelling example, CAuWS would work as well on any other CPS meeting the requirements for precomputation. Likewise, more complex environments result in more physical constraints, creating a more challenging scheduling problem. CAuWS can represent simple environments too. Many real systems will often have similar numbers of constraints, though perhaps with less drastic failures.

Lastly, we demonstrate the variety of constraints CAuWS supports with a set of small, synthetic examples.

4.4 Workload Profiling

The time, energy, and power use of the computational tasks must be predetermined for CAuWS. As the compute platform, we picked NVIDIA's Jetson Xavier AGX [1] and NX. They both provide two accelerators: GPU (low-latency) and DLA (low-power). We ran and profiled a variety of neural networks (NNs) supported by TensorRT [68] on both on GPU and DLA. Data was collected with the offline profiler *IProfiler* over 5000 iterations, excluding the first 1000 iterations of the warm-up period. During operation, the drone would also repeatedly be running these networks and not need warm-up. We profile ORB-SLAM [69] on the CPU with one to four reserved cores using the C++ chrono library. Profiling data is fed to CAuWS via a seperate file listing latency, power, and energy consumption values.

4.5 Case Study 1: Environment-Limited Search and Rescue

In this section, we first detail our simulation environment and the constraints this induces. We then detail the three constraints we use in the case study and how we integrate them into CAuWS. We finally discuss how we ensure that the constrained optimization problems that CAuWS generates remain linear, and thus effectively solvable.

The first simulation is a search-and-rescue task with a quadcopter (3DR Iris) in a burning house. The drone must explore the house to monitor fires and discover those in need of help. There are also industrial applications in hot environments that may present similar constraints. Such a drone is useful as it can explore without putting firefighters at risk. Fires [70] and built environments can also disrupt radio communication, and so autonomy is often necessary for such situations.

For this task, the drone has a camera and an Nvidia Xavier AGX SOC, notably containing an 8-core ARM CPU, a Volta based GPU, and a Deep Learning Accelerator (DLA).

The simulation follows a predefined route through a hallway corner with a heat source. Fig. 4.1 provides screenshots of this scenario from the simulation environment.

The drone must run the following tasks to complete its objective: (1) An image identification network to identify humans in need of rescue in order to contact help, (2) an image segmentation network to parse the environment into walls, doorways, and fires, and (3) Simultaneous Localization and Mapping (SLAM) to know its position in the environment (which can also correlate with the segmentation). These tasks were profiled offline on the AGX in terms of computation time and energy. This creates a workload with dependencies that CAuWS must then schedule.

Once the drone has the output from these tasks, it can plan its future actions on a solo CPU. We neglect this task as the time and energy it contributes is negligible, especially when run in parallel to the more intensive task. As CAuWS pre-computes schedules before operation, the cost of selecting the appropriate

schedule is also negligible.

4.5.1 Constraint Linearization

MILP solvers support linear constraints; however, physical dynamics often contain nonlinearities. For example, velocity v appears in the terms $vt^{\langle \text{end} \rangle}$ and v^2 in Sec. 4.2. Importantly, velocity is not a "decision variable"—i.e., not an output of CAuWS. Following the algorithm we proposed in subsubsection 3.9.4, we query CAuWS to find schedules for a given v, so v^2 is also constant for that query. This lets us query multiple linear problems to find (possibly nonlinear) dividing borders in Fig. 3.5.

Additionally, the steady state heat equation is used to avoid an exponential relationship between $\sum E_{op}$ and $t^{\langle \text{end} \rangle}$.

4.5.2 Results Overview

The drone's travel results in ambient temperatures ranging from 27 to 73 $^\circ C$, velocities up to $9.73 \frac{m}{s}$, and obstacle distances as low as $0.3m$.

A trace of this simulation, shown in Fig. 4.1, records temperature, velocity, and obstacle distance. These values were fed into CAuWS, which successfully finds the best schedules to satisfy the constraints and optimization objective. For CAuWS to accurately consider physical conditions in the simulation, there must be accurate representations in the AuWL file. The various constants were determined either from the 3DR Iris specification or experimentally from simulation as described in Sec. 4.2. The resulting AuWL file is given in Fig. 4.2.

4.5.3 Resulting Schedules

After profiling data is integrated, the solver results in the set of schedules shown in Fig. 3.5. As long as the drone stays within the scheduleable region shown, CAuWS can find a schedule predicted to meet the constraints. The path taken is drawn in Fig. 4.1, with the different schedules colored.

The drone adapts to the varying conditions with these schedules. Along this path, the drone goes around the corner, skirting the wall, before passing close to the fire. The corner reduces obstacle distance and required latency (schedule 1). When near the heat source, the more energy efficient schedule must be chosen (schedule 3). Otherwise, it defaults to the in-between schedule 2.

Past schedulers, which do not consider physical constraints, could only choose one of these possible schedules and would violate physical constraints for at least 25% of the length of the path. If the path went significantly closer to the wall or fire there would be no possible schedules regardless of scheduler.

```
model case_study_one {
  cnstrnt (= maxAcc (/ mass
    (* batteryPow newtonsPerWatt)) )
  cnstrnt (= stopDist (+ (* $svel $svel TIME )
    (/ (* $svel $svel $svel $svel)
      (* 2.0 maxAcc))) )
  cnstrnt (< stopDist $distToObs)
  cnstrnt (= velPow (* velocityVsPowerConstant $svel))
  cnstrnt (> batteryPow (+ POWER velPower idlePower))
  cnstrnt (= tmpDelta (- maxTemp $ambTemp))
  cnstrnt (= maxTempOut (* TIME kConductivity tmpDelta))
  cnstrnt (> maxTempOut (* HEAT tempPerJoule) )
  objective  (* -1 (+ (* 0.5 HEAT)
    (* 1.5 TIME) (* -2 POWER)))
  data camera, position
  data object_bounding_boxes, hazard_segmentation
  op resnet {in=camera;out=object_bounding_boxes}
  op fcn {in=camera;out=hazard_segmentation}
  op slam {in=camera;out=position}
}
```

Figure 4.2 The AuWL file corresponding to the first case study. Defines the control flow graph and places constraints on heat generation, power consumption, and latency. See Sec. A for the full file

.

4.5.4 Predicted vs. Simulated Results

These schedules were then ran on the Xavier AGX to follow the trace, with each schedule running multiple times in a row over the course of the drone's path. This resulted in a total execution time of $9.96s$ and energy of $84.5J$, compared to the predicted time of $9.19s$ and energy of $96.7J$. These errors of 8% and 14% (respectively) are very consistent. The consistency would easily allow a safety margin to be added to time constraints. The errors would likely be more permanently resolved by considering caching, contention, and other effects that occur when the operations are not isolated.

4.5.5 Potential Future Work

While this scenario demonstrates the important tradeoff between latency and energy while considering physical constraints, there are other considerations that could be interesting. In this specific example scenario, the power constraint does not end up impacting schedule choice. Throttling would be another option to consider in order to avoid overheating, though CAuWS is capable of representing discrete throttling values. Likewise, it may be possible to subdivide neural network layers for finer gradients of schedules.

Some considerations also need a measure of quality we do not have. It would be possible to consider smaller networks to reduce time and latency at the cost of accuracy. However, evaluating this requires some measurement of how these choices impact the quality or success rate of the result. Examining how CAuWS

could be used to consider quality remains an area of future work.

4.6 Case Study 2: Discovery & Tracking

We also demonstrate CAuWS's versatility in mapping physical constraints of CPS to computational scheduling with two additional simulations, where an aerial drone must discover and follow another aerial adversary. In both scenarios, the adversary is represented with a pre-defined velocity curve that our drone follows. The device used is also the NX instead of the AGX. These experiments demonstrate that CAuWS can handle different modes of operation with in addition to dynamic environments pre-computation, expanding the CPS CAuWS is applicable to.

4.6.1 Criteria 1: Power Limits

As explained in detail in Sec. 4.2, power in a drone is shared between computation and the actuation (rotors), and the total power a battery can provide is limited. Thus, rapid flight may require the NN to run on the more power efficient DLA instead of the GPU. We test this scenario, simulated with the PX4 simulator [71], under the following objective and constraints in the AuWL input in Fig. 4.3:

```
constraint (= drone-power (+ rotor-power idle-power))
constraint (< (+ drone-power POWER) 402))
objective  (- TIME)
```

Figure 4.3 An AuWL snippet used in case study two. Minimizes latency while maintaining a power limit.

Fig. 4.4 is a timeline of resulting power consumption. The adversary may choose a velocity to which the drone must adapt, and greater adversary velocity requires greater drone velocity and power. While computation uses only a fraction of physical power, with a tight limit the schedule must reduce power by using the efficient DLA to complete the drone's mission.

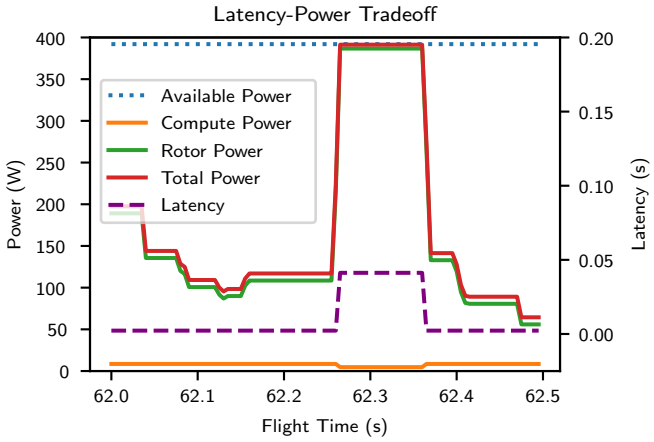

Figure 4.4 Power consumption in a pursuit flight. CAuWS balances power and latency under total power limits. Power values are the max over a .05s window to conservatively satisfy power limits.

4.6.2 Criteria 2: Multiple Modes and Latency Limits

The drone discovery and tracking scenario consists of two modes: looking for an adversary and following the adversary. These two modes impose different scheduling criteria. CAuWS is still able to operate in different modes with precomputation, despite its static nature. When looking for an adversary, the drone must minimize power consumption to maximize flight time. Once the drone detects and begins to follow the adversary, the schedule requirements change. First, the drone has a hard constraint on computation time to ensure the adversary does not leave the image between frames. Second, the drone must maximize tracking accuracy (trade-offs between efficient and accurate networks), minimize its power use (improving flight time), and minimize computation time. Assuming the drone can safely take longer to perform object detection, the quality of the object detection becomes most important. We represent this scenario using following AuWL constraints and a linear combination of objectives in Fig. 4.5:

```
constraint (= minimum-latency (/ 0.035 adv-velocity))
constraint (< TIME minimum-latency)
objective  (+ ACCURACY (/ POWER -1000) (/ TIME -10000))
```

Figure 4.5 An AuWL snippet used in case study two. Prioritizes accuracy, then power, then time

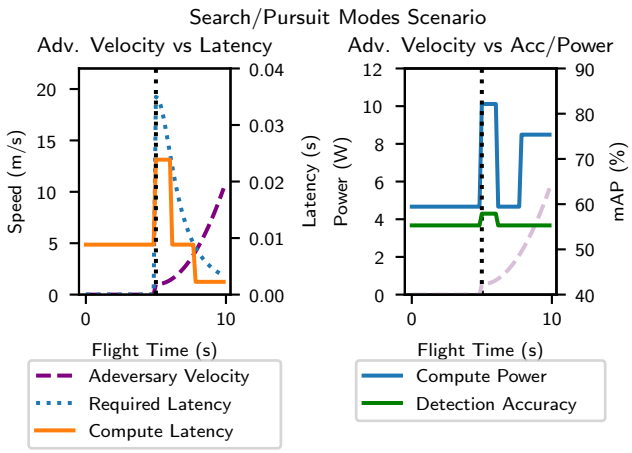

Figure 4.6 CAuWS can create schedules covering multiple operation "modes" in a simulated flight, enforcing different constraints for each. This example includes latency, network size, and power in a simulated pursuit scenario where the required computation speed increases when the adversary speeds up after detection.

Fig. 4.6 shows the results of this multi-mode scenario. CAuWS successfully chooses the lowest power option leading up to the encounter (shown with vertical dashed lines). After the encounter, the necessary latency is maintained to account for the adversary's velocity optimizing the other values in priority order (accuracy, then power).

4.7 Comparison with the State-of-the-Art

To our knowledge, there are no related works that make the same full set of considerations as CAuWS. They all differ through some combination of:

- Not considering scheduling
- Not considering scheduling for *heterogeneous* SoC's
- Not considering physical constraints
- Considering queues of waiting tasks instead of a known CFG

As such, a direct comparison with previous work becomes difficult. Instead we compare aspects of CAuWS against two bodies of work:

- The F-1 technique [12, 72] which, while it does not produce schedules, does select SoC's [12] and generated domain-specific SoC's [72] from a physically constrained Pareto frontier. The single factor, stopping distance, that F-1 considers is also a constraint in our case study.
- Π-RT [73], a scheduling approach that can produce schedules for heterogeneous SoC's. It provides methods that attempt to either reduce latency or energy use. This technique, while it cannot consider arbitrary CFG's, can consider tasks at a similar granularity to CAuWS. Unlike CAuWS, Π-RT cannot adhere to

physical constraints, requires tuning of hyper-parameters, and cannot automatically adapt its scheduling approach to varying conditions.

4.7.1 F-1: Physical Constraints

State-of-the-art constraint based schedulers do exist, but do not use *physical* constraints—instead the constraints represent dependencies, latencies, and similar. The *F-1* technique from [12, 72] is the closest comparison that considers *physical* constraints in relation to computation systems. However, F-1 does not make scheduling decisions, focusing on other applications. CAuWS is the first work we are aware of that combines both constraint scheduling and physical considerations. F-1 proposes a mathematical way of relating a single stopping-distance limitation to choosing an SoC to include on a drone, based on the tradeoffs between SoC weight, SoC latency, rotor weight, and rotor power.

CAuWS differs by considering *scheduling* on *multiple* parameters *and* devices. While F-1 only considers velocity, CAuWS considers constraints on heat, velocity, on power. This expansion is non-trivial as the optimum of the roofline models is a maximum over all constraints, while multiple dimensions adds tradeoffs between values such that increasing one value could violate a constraint on another. CAuWS can also represent the relation in [12] with the AuWL in Fig. 4.7:

```
(= distance 10)
(> distance  (+ (* $vel TIME) (* $vel $vel AMAXINV)))
```

Figure 4.7 AuWL snippet relating maximum acceleration (depending on processor choice) to the stopping distance, for comparison to F-1.

This AuWL snippet has a constant based on acceleration (the value $\frac{1}{2a_{max}}$, AMAXINV in the AuWL) as an additional "resource" produced by a processor choice operation. When searched with the binary partitioning algorithm (Listing 3.1), this will divide velocity into a valid and invalid region, like Fig. 3.5—the border between these being the maximum velocity.

4.7.2 Π-RT: Task-Level Scheduling for Heterogeneous SoCs

While many works exist that can schedule for heterogeneous SoC's, Π-RT [73] is the closest comparison we are aware of to CAuWS, as Π-RT and CAuWS share the same inputs (a set of tasks and profiling data) and outputs (queues assigning tasks to processors). There are three ways in which CAuWS has more capabilities than Π-RT, also shown in Fig. 4.8:

- CAuWS finds globally optimal solutions without tuning. Previous works such as Π-RT that perform heterogeneous scheduling often use heuristics to avoid the NP-complete complexity. Admittedly, this does mean they can handle much larger problems that CAuWS. However, this can result in non-optimal schedules, and can require fine-tuning of heuristic parameters. This can be unnecessary as many CPS applications, especially those adhering to our assumptions, have relatively few tasks. CAuWS, without tuning, finds the most optimal schedule within the constraints Fig. 4.8. Without any constraints CAuWS is guaranteed by the MILP solver objective to find of of the Pareto-optimal schedules.
- CAuWS produces schedules adhering to physical constraints while Π-RT does not consider physical constraints. Additionally, CAuWS can find schedules that Π-RT cannot, which are sometimes necessary to fulfill the constraints. Shown in Fig. 4.8 are all possible schedules from choosing Π-RT's hyper-parameters (queue weights) and energy or latency first approach—none lie within the valid region.
- CAuWS automatically adapts schedules to these constraints. While Π-RT provides energy and latency first approaches, these have to manually selected ahead of time based on knowledge of the task. CAuWS produces multiple schedules that can automatically adapt to these conditions, seen with the different graphs in Fig. 4.8.

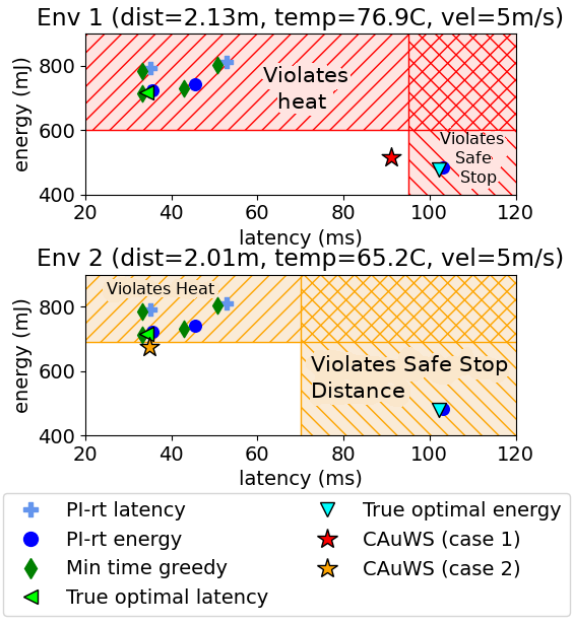

Figure 4.8 A comparison of the possible schedules the two approaches of Π-RT [73], a time-first greedy algorithm, and CAuWS generate. The two environments induce different time and energy constraints, which CAuWS can adjust for. All of the possible schedules from Π-RT and the greedy scheduler lie outside the constraints for these examples. Π-RT requires some parameter tuning, hence the multiple points.

CHAPTER 5

PROPOSED INTEGRATION OF MOTION PLANNING

5.1 Problem Description

Robots, among other systems, need to find paths between desired configurations. This problem of finding paths is non-trivial as the shortest path between two points will often run into obstacles. This is further complicated when the problem is not only finding a path between two points, but finding the best path (in terms of length, time, energy, or other resources).

It is possible to include scheduling decisions in motion planning, as the relation between them works both ways. With a way to programatically determine the best schedule for points in a motion plan, it is possible to determine motion plans by considering what points allow which schedules.

By incorporating CAuWS into motion planning, it is possible to find more efficient paths compared to a planner that ignores scheduling considerations. By having CAuWS as a complete system that allows us to query possible schedules, CAuWS can be incorporated as a subsystem in motion planning. We suggest a way CAuWS could thus be incorporated into a Motion Planner to find more efficient schedules, and to find plans in a larger variety of environments.

The motion planning problem can more formally be stated for our specific case: For a robot with an n dimensional configuration space, there is a configuration space $\mathcal{C}$ describing the state (e.g. position, orientation) of the robot. $\mathcal{C}$ has a free region $\mathcal{F}$ and obstacle region $\mathcal{O}$ that do not overlap. Motion planning seeks to find a path between two configurations that lie entirely within $\mathcal{F}$.

CAuWS can then be used to invalidate some motions when sampling. $\mathcal{O}$ is composed both of configurations that collide with physical obstacles, and configurations that are impossible to schedule for. The robot can not be in configurations from either of these, and CAuWS allows us to check whether a motion contains configurations with no valid schedules, or to determine the best schedule otherwise. While the framework of motion planning supports incorporating scheduling validity into configuration validity in such an approach, no past approaches have evaluated doing so.

We describe how such considerations could be implemented within a sampling-based motion planning framework, such as those provided in [49].

Velocity constraints (as proposed in Sec. 4.2) also add another complication, as velocity must now also be tracked as part of the configuration—potentially requiring the use of, e.g., a control-space planner.

5.2 Potential Scenario

To motivate the usefulness such a technique may be able to provide, it helps to consider the same environment detailed in the experiments for CAuWS (Sec. 4.2). While our proposed technique may be able to consider other situations, this scenario helps demonstrate a gap in the capability of current planners.

The scenario is a search-and-rescue mission during a fire. While the past sections investigated the impact this scenario has on scheduling, this environment presents two types of hazards that affect motion planning decisions too—obstacles (walls, furniture) and heat sources (fires). Both hazards have a choice in how to address them: either move around them with a longer path, or choose a different, worse schedule when close enough that the unconstrained schedule would be invalid. Instead of choosing schedules that adhere to constraints along a path, it is possible to choose a path that adheres to the schedules.

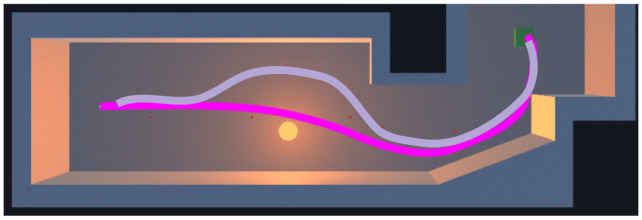

Figure 5.1 An example of two paths where the longer path could potentially take less time to fly.

As an example, consider the two paths in Fig. 5.1. The bottom path approaches the heat source more closely, which requires lower energy (but higher latency) schedules. The upper path instead stays far enough away to use the lower latency schedules as necessary. This choice presents a tradeoff to make when planning. While the bottom path is physically shorter, it also must be driven slower. Depending on the exact path in question, this can result in a longer total time and energy consumption. There are other situations when the bottom path still takes less time, however, and motion planning can choose between them.

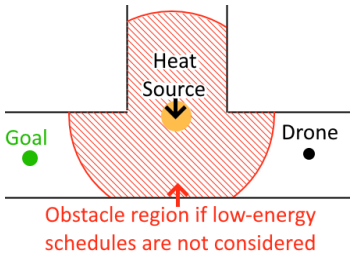

Figure 5.2 Some paths can be infeasible without being able to reduce the computational energy use with schedules.

Previous works that do not consider scheduling during planning can not consider the tradeoffs between paths and available schedules. One of three approaches would have to be taken to have a comparable result,

though these ideas could be applied with any planner:

- The lower-latency schedule was used for planning the entire path, excluding the robot from hot areas and possibly making some motions infeasible.
- The lower-energy schedule could be used for planning the entire path, forcing the robot to move slower than optimal in cold areas.
- During planning, the planner could ignore schedules—then when executing, it would use real-time scheduling approaches (such as [52–54]) and adjust velocity accordingly to the latency. In this case, what the planner assumes to be the shortest path may go through a hot area and slow down the robot after, when avoiding the hot area but driving faster would take less time.

An approach that considers schedules *during* planning could improve over these approaches.

Choosing the lower energy schedule avoids this problem, but limits the maximum speed that can be driven, which can lead to longer paths.

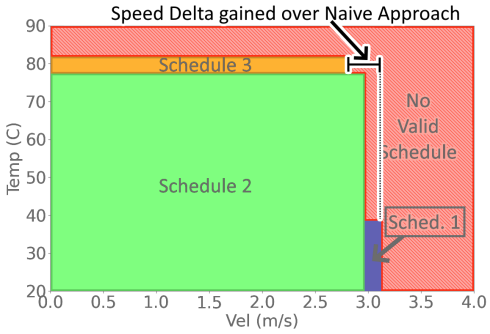

Figure 5.3 The amount of velocity gained by switching schedules is quantifiable, and while small, is still significant enough to improve path travel times.

There are some cases where considering these schedules will improve path times given the velocity differences shown in Fig. 5.3. This is demonstrable analytically—as an extreme example, consider a planning problem with a choice of two hallways, with one hallway kept just above the temperature requiring Schedule 3. This hallway must then be travelled at a slower speed—if the other hallway is longer, but not by a greater proportion than the speed delta, it will still be faster. While such a scenario is unlikely, many more scenarios should show slight improvements. In addition to such improvements, there are scenarios where these considerations are necessary to be able to create safe motion plans—e.g. a single hallway completely blocked by a hot temperature.

5.3 Proposed Methodology

We propose using sampling-based control-space planners (sometimes known as planning with differential constraints) to satisfy these requirements. We propose using an RRT [47, 48] approach. To summarize how the proposed planning approach (adapted from the RRT Control-Space Planner in OMPL [49]) considers schedules:

```
Start with a starting state S and a goal state G
While S is not connected to G:
    Randomly sample a target state R from C
    Find the nearest state N connected to S
    Calculate a physically−possible control C
        to move in the direction of R
    Solve the differential dynamics of the robot
        to apply C to R for timestep(s),
        obtaining new state(s) Q
    If Q intersects with obstacles:
        Move to the next iteration
    ----------------------------------
    If Q lies in the unschedulable region:
        Move to the next iteration
    ----------------------------------
    Attach Q to N
    If Q is near to G:
        Attach G to Q
```

Listing 5.2: RRT* with schedules

This algorithm adapts a control space planner in order to include velocity. As an RRT planner, the main loop expands a tree of possible motions from the start until it finds a motion that reaches the goal. *Our key change that differs from existing control-space RRT approaches is the highlighted section, which checks that sampled points have a valid schedule before adding them to the tree.*

Within this loop, it chooses a direction to expand by sampling a random state R from the the configuration space $\mathcal{C}$. It must then choose a state from which to move towards R, by selecting the nearest state N in the current tree. The metric of "nearest" may be more complicated than euclidean distance (L-2 Norm) if velocity or travel time are considered.

As a control-space planner, it then determines controls C to move roughly in the direction of R. Some pairings of R and N may not have a control that moves directly between them. R and N may not have a feasible motion given robot dynamics, while the motion between other pairs may pass through invalid regions or be subject to noise. This step to calculate C accounts for these possibilities.

Next, C is propagated an amount of time to determine a new state Q (alternatively, it can be propagated for multiple timesteps to obtain multiple new states). Q must then be checked for validity (presence in $\mathcal{F}$ or $\mathcal{O}$) which considers both physical obstacles and *possible schedules*.

Assuming Q is valid, it is added to the tree with it's parent as N. An additional check is then made to see if G is reachable—if so, then a complete path has been found.

To adapt this to an RRT* planner [50] from an RRT planner, a check must also be made when sampling new points to see if they improve any existing paths—otherwise the first path to reach the goal may be inefficient. This also requires tracking current path cost from S to all states.

5.4 Conclusion

In this work, we presented CAuWS, a system for producing schedules for heterogeneous systems. These schedules adhere to physical constraints, and can adapt to changing environments by precomputing a set of schedules. We the demonstrate CAuWS with a case study, showing how it chooses schedules depending on the different physical conditions of a simulated drone during its flight. Lastly, we propose a method to incorporate these decisions into motion planning in order to reduce the energy or total time of a motion.

REFERENCES

[1] NVIDIA. Ai-powered autonomous machines at scale — nvidia jetson agx xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/, . (Accessed on 4/21/2022).

[2] Tesla. Artificial intelligence & autopilot. https://www.tesla.com/AI. (Accessed on 11/20/2021).

[3] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[4] Kyle L. Spafford and Jeffrey S. Vetter. Aspen: A domain specific language for performance modeling. In *SC*, 2012.

[5] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[7] Nathan R Tallent and Adolfy Hoisie. Palm: easing the burden of analytical performance modeling. In *ICS*, 2014.

[8] Arnamoy Bhattacharyya and Torsten Hoefler. Pemogen: Automatic adaptive performance modeling during program runtime. In *PACT*, 2014.

[9] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *ASPLOS*, 2018.

[10] B. Yu, W. Hu, L. Xu, J. Tang, S. Liu, and Y. Zhu. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *MICRO*, 2020.

[11] Edward A Lee. The past, present and future of cyber-physical systems: A focus on models. *Sensors*, 15 (3):4837–4869, 2015.

[12] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. The sky is not the limit: A visual performance model for cyber-physical co-design in autonomous machines. *IEEE CAL*, 2020.

[13] Ramyad Hadidi, Bahar Asgari, Sam Jijina, Adriana Amyette, Nima Shoghi, and Hyesoon Kim. Quantifying the design-space tradeoffs in autonomous drones. In *ASPLOS*, pages 661–673, 2021.

[14] 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.

[15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νz-an optimizing SMT solver. In *TACAS*, volume 15, pages 194–199, 2015.

[16] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2022. URL http://www.gurobi.com.

[17] Kevin J Brown, Arvind K Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, 2011.

[18] K. Li, X. Tang, and K. Li. Energy-efficient stochastic task scheduling on heterogeneous computing systems. *IEEE Trans. on Parallel and Distributed Systems*, 25(11):2867–2876, 2014. doi: 10.1109/TPDS.2013.270.

[19] X. Mei, X. Chu, H. Liu, Y. Leung, and Z. Li. Energy efficient real-time task scheduling on cpu-gpu hybrid clusters. In *INFOCOM*, 2017.

[20] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *ISCA*, 2012.

[21] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker. Scalable thread scheduling and global power management for heterogeneous many-core. In *PACT*, 2010.

[22] H. Arabnejad and J. G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE TPDS*, 2014.

[23] Dominik Grewe and Michael FP O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Intl. conference on compiler construction*, pages 286–305. Springer, 2011.

[24] Minhaj Ahmad Khan. Scheduling for heterogeneous systems using constrained critical paths. *Parallel Computing*, 38(4-5):175–193, 2012.

[25] Seren Soner and Can Özturan. Integer programming based heterogeneous cpu–gpu cluster schedulers for slurm resource manager. *Journal of computer and system sciences*, 81(1):38–56, 2015.

[26] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *CGO*, 2019.

[27] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *TACO*, 2013.

[28] Mohammad I Daoud and Nawwaf Kharma. A hybrid heuristic–genetic algorithm for task scheduling in heterogeneous processor networks. *JPDC*, 2011.

[29] Zishen Wan, Aqeel Anwar, Yu-Shun Hsiao, Tianyu Jia, Vijay Janapa Reddi, and Arijit Raychowdhury. Analyzing and improving fault tolerance of learning-based navigation systems. In *DAC*, 2021.

[30] Sekhri Larbi and Slimane Mohamed. Modeling the scheduling problem of identical parallel machines with load balancing by time petri nets. *Intl. Journal of Intelligent Systems & Applications*, 7(1), 2014.

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

[32] Haitao Zhang and Feiyue Wang. A review of petri net based modeling and verification for embedded real-time systems. In *IDETC-CIE*, 2005.

[33] I. Demongodin and N. T. Koussoulas. Differential petri nets: representing continuous systems. *IEEE Trans. on Automatic Control*, 1998.

[34] Zhigang Hu, Xiangtao Jiang, and Jianbiao He. Performance analysis technique for fixed priority scheduling with hybrid real-time transactions. In *2008 Intl. Conference on Information and Automation*, pages 509–513, 2008. doi: 10.1109/ICINFA.2008.4608053.

[35] Carlo Ghezzi, Dino Mandrioli, Sandro Morasca, and Mauro Pezze. A unified high-level petri net formalism for time-critical systems. *IEEE Transactions on software engineering*, 17(2):160, 1991.

[36] Ion Dan Mironescu and Lucian Vinţan. Coloured petri net modelling of task scheduling on a heterogeneous computational node. In *ICCP*, 2014.

[37] M. Naedele. Petri net models for single processor real-time scheduling. 1998.

[38] Jeffrey J. P. Tsai, S Jennhwa Yang, and Yao-Hsiung Chang. Timing constraint petri nets and their application to schedulability analysis of real-time system specifications. *IEEE transactions on Software Engineering*, 1995.

[39] Luis Alejandro Cortés, Petru Eles, and Zebo Peng. A petri net based model for heterogeneous embedded systems. In *Proc. NORCHIP Conference*, pages 248–255, 1999.

[40] Henry A. Kautz and Bart Selman. Planning as satisfiability. *ECAI*, 92:359–363, 1992.

[41] Neil T. Dantam, Zachary K. Kingston, Swarat Chaudhuri, and Lydia E. Kavraki. An incremental constraint-based framework for task and motion planning. *IJRR*, 37(10):1134–1151, 2018.

[42] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[43] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[44] Steven M La Valle. Motion planning. *IEEE Robotics & Automation Magazine*, 18(2):108–118, 2011.

[45] Enric Galceran and Marc Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous systems*, 61(12):1258–1276, 2013.

[46] Mohamed Elbanhawi and Milan Simic. Sampling-based robot motion planning: A review. *Ieee access*, 2: 56–77, 2014.

[47] Steven M LaValle et al. Rapidly-exploring random trees: A new tool for path planning. Technical Report 98-11, Iowa State University, Ames, IA, 1998.

[48] 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.

[49] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. doi: 10.1109/MRA.2012.2205651. https://ompl.kavrakilab.org.

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

[51] Iram Noreen, Amna Khan, and Zulfiqar Habib. Optimal path planning using rrt* based approaches: a survey and future directions. *International Journal of Advanced Computer Science and Applications*, 7 (11), 2016.

[52] Guoqi Xie, Gang Zeng, Zhetao Li, Renfa Li, and Keqin Li. Adaptive dynamic scheduling on multifunctional mixed-criticality automotive cyber-physical systems. *IEEE Transactions on Vehicular Technology*, 66(8):6676–6692, 2017.

[53] Aporva Amarnath, Subhankar Pal, Hiwot Tadese Kassa, Augusto Vega, Alper Buyuktosunoglu, Hubertus Franke, John-David Wellman, Ronald Dreslinski, and Pradip Bose. Heterogeneity-aware scheduling on socs for autonomous vehicles. *IEEE Computer Architecture Letters*, 20(2):82–85, 2021.

[54] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011.

[55] Zhiyu Liu, Jin Dai, Bo Wu, and Hai Lin. Communication-aware motion planning for multi-agent systems from signal temporal logic specifications. In *2017 American Control Conference (ACC)*, pages 2516–2521. IEEE, 2017.

[56] Yuan Yan and Yasamin Mostofi. To go or not to go: On energy-aware and communication-aware robotic operation. *IEEE Transactions on Control of Network Systems*, 1(3):218–231, 2014.

[57] Yi Guo and Lynne E Parker. A distributed and optimal motion planning approach for multiple mobile robots. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*, volume 3, pages 2612–2619. IEEE, 2002.

[58] Nuwan Ganganath, Chi-Tsun Cheng, and K Tse Chi. A constraint-aware heuristic path planner for finding energy-efficient paths on uneven terrains. *IEEE transactions on industrial informatics*, 11(3): 601–611, 2015.

[59] Tanmoy Kundu and Indranil Saha. Energy-aware temporal logic motion planning for mobile robots. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8599–8605. IEEE, 2019.

[60] Yazz Warsame, Stefan Edelkamp, and Erion Plaku. Energy-aware multi-goal motion planning guided by monte carlo search. In *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, pages 335–342. IEEE, 2020.

[61] Soumya Sudhakar, Sertac Karaman, and Vivienne Sze. Balancing actuation and computing energy in motion planning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4259–4265. IEEE, 2020.

[62] Sean Murray, Will Floyd-Jones, Ying Qi, Daniel J Sorin, and George Dimitri Konidaris. Robot motion planning on a chip. In *Robotics: Science and Systems*, volume 6, 2016.

[63] Sabrina M. Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. Robomorphic computing: A design methodology for domain-specific accelerators parameterized by robot morphology. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 674–686, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446746. URL https://doi.org/10.1145/3445814.3446746.

[64] Adnane Saoud. *Compositional and efficient controller synthesis for cyber-physical systems*. PhD thesis, Université Paris-Saclay (ComUE), 2019.

[65] Qi Zhu and Alberto Sangiovanni-Vincentelli. Codesign methodologies and tools for cyber–physical systems. *Proceedings of the IEEE*, 106(9):1484–1500, 2018. doi: 10.1109/JPROC.2018.2864271.

[66] Christos G Cassandras and Stephane Lafortune. *Introduction to discrete event systems*, volume 2. Springer, 2008.

[67] Jussi Rintanen. Engineering efficient planners with SAT. In *ECAI*, pages 684–689, 2012.

[68] NVIDIA. Tensorrt. . URL https://developer.nvidia.com/tensorrt. (Accessed on 4/21/2022).

[69] Raúl Mur-Artal and Juan D. Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017. doi: 10.1109/TRO.2017.2705103.

[70] Jonathan Alexander Boan. *Radio propagation in fire environments.* PhD thesis, 2009.

[71] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. *Robot Operating System (ROS): The Complete Reference (Volume 1)*, chapter RotorS—A Modular Gazebo MAV Simulator Framework. Springer International Publishing, 2016.

[72] Aleksandra Faust, David Brooks, Gu-Yeon Wei, Kshitij Bhardwaj, Paul Whatmough, Srivatsan Krishnan, Vijay Janapa Reddi, and Zishen Wan. Automatic domain-specific soc design for autonomous unmanned aerial vehicles. 2022.

[73] Liu Liu, Jie Tang, Shaoshan Liu, Bo Yu, Yuan Xie, and Jean-Luc Gaudiot. Pi-rt: A runtime framework to enable energy-efficient real-time robotic vision applications on heterogeneous architectures. *Computer*, 54(4):14–25, 2021. doi: 10.1109/MC.2020.3015950.

APPENDIX

AUWL DETAILS

A.1 Full AuWL File

```
model car_example {
  depleted HEAT
  claimed  POWER
  cnstrnt (= newtonsPerWatt 0.020979)
  cnstrnt (= velocityVsPowerConstant 7.602631123)
  cnstrnt (= mass 1.5)
  cnstrnt (= batteryPower 1800)
  cnstrnt (= maxTemp 85)
  cnstrnt (= kConductivity .00559)
  cnstrnt (= tempPerJoule 0.01685714286)
  cnstrnt (= idlePower 700.7)
  cnstrnt (= maxAcc (/ mass
    (* batteryPower newtonsPerWatt)) )
  cnstrnt (= stopDist (+ (* (* $svel $svel)
      (* TIME 0.001))
    (/ (* $svel $svel $svel $svel)
      (* 2.0 maxAcc))) )
  cnstrnt (< stopDist $distToObs)
  cnstrnt (= velPower   (* velocityVsPowerConstant $svel))
  cnstrnt (= actPower (+ velPower idlePower))
  cnstrnt (< actPower (- batteryPower POWER))
  cnstrnt (= maxTempOut (* 0.001 TIME
    kConductivity (- maxTemp $ambTemp)))
  cnstrnt (< (* HEAT 0.001 tempPerJoule)
    maxTempOut)
  cnstrnt (< TIME 120.0)
  objective  (* -1 (+ (* 0.5 HEAT)
    (* 1.5 TIME) (* -2 POWER)))
  data camera_a, camera_b, lidar, position
  data object_bounding_boxes, hazard_bounding_boxes
  op resnet {in=camera_a;out=object_bounding_boxes}
  op fcn {in=camera_b;out=hazard_bounding_boxes}
  op slam {in=lidar;out=position}
}
```

A.2 AuWL Grammar

$$
\begin{array}{rcl}
\langle \text{program} \rangle & \rightarrow & \text{“model”} \langle \text{NAME} \rangle \text{“\{”} \langle \text{param\_list} \rangle^* \langle \text{function} \rangle^* \langle \text{data\_item} \rangle^* \langle \text{operation} \rangle^* \text{“\}”} \\
\langle \text{param\_list} \rangle & \rightarrow & \text{“param”} \langle \text{params} \rangle \\
\langle \text{params} \rangle & \rightarrow & \langle \text{param} \rangle \\
 & | & \langle \text{param} \rangle \text{“,”} \langle \text{params} \rangle \\
\langle \text{param} \rangle & \rightarrow & \langle \text{NAME} \rangle \text{“=”} \langle \text{expression} \rangle \\
\langle \text{function} \rangle & \rightarrow & \text{“fun”} \langle \text{NAME} \rangle \text{“(”} \langle \text{name} \rangle \text{“)”} \text{“=”} \langle \text{expression} \rangle \\
\langle \text{expression} \rangle & \rightarrow & \langle \text{term} \rangle \langle \text{ADD\_OP} \rangle \langle \text{term} \rangle^* \\
\langle \text{term} \rangle & \rightarrow & \langle \text{factor} \rangle \langle \text{MUL\_OP} \rangle \langle \text{factor} \rangle^* \\
\langle \text{factor} \rangle & \rightarrow & \langle \text{NAME} \rangle \text{“(”} \langle \text{expression} \rangle \text{“,”} \langle \text{expression} \rangle^* \text{“)”}? \\
 & | & \langle \text{NUMBER} \rangle \\
 & | & \text{“(”} \langle \text{expression} \rangle \text{“)”} \\
\langle \text{data\_list} \rangle & \rightarrow & \langle \text{data\_item} \rangle \\
 & | & \langle \text{data\_item} \rangle \langle \text{data\_list} \rangle \\
\langle \text{data\_item} \rangle & \rightarrow & \text{“data”} \langle \text{NAME} \rangle \langle \text{dimension} \rangle^* \langle \text{tags} \rangle? \\
\langle \text{dimensions} \rangle & \rightarrow & \langle \text{dimension} \rangle \\
 & | & \langle \text{dimension} \rangle \langle \text{dimensions} \rangle \\
\langle \text{dimension} \rangle & \rightarrow & \text{“[”} \langle \text{NAME} \rangle \text{“]”} \\
\langle \text{operation\_list} \rangle & \rightarrow & \langle \text{operation} \rangle \\
 & | & \langle \text{operation} \rangle \langle \text{operation\_list} \rangle \\
\langle \text{operation} \rangle & \rightarrow & \text{“op”} \langle \text{NAME} \rangle \langle \text{tags} \rangle \\
\langle \text{tags} \rangle & \rightarrow & \text{“\{”} \langle \text{tag\_list} \rangle \text{“\}”} \\
\langle \text{tag\_list} \rangle & \rightarrow & \langle \text{tag} \rangle \\
 & | & \langle \text{tag} \rangle \text{“;”} \langle \text{tag\_list} \rangle \\
\langle \text{tag} \rangle & \rightarrow & \langle \text{NAME} \rangle \text{“=”} \langle \text{values} \rangle \\
\langle \text{values} \rangle & \rightarrow & \langle \text{value} \rangle \\
 & | & \langle \text{value} \rangle \text{“,”} \langle \text{values} \rangle
\end{array}
\tag{A.1}
$$

Figure A.1 The basic AuWL grammar.