Scheduling for Cyber-Physical Systems with Heterogeneous Processing Units under Real-World Constraints

Justin McGowen
Colorado School of Mines
Golden, CO, USA
jmcgowen@mines.edu

Ismet Dagli
Colorado School of Mines
Golden, CO, USA
ismetdagli@mines.edu

Neil T. Dantam
Colorado School of Mines
Golden, CO, USA
ndantam@mines.edu

Mehmet E. Belviranli
Colorado School of Mines
Golden, CO, USA
belviranli@mines.edu

ABSTRACT

Cyber-physical systems (CPS) such as robots and self-driving cars pose strict physical requirements to avoid failure. The scheduling choices impact these requirements. This presents a challenge: How do we find efficient schedules for CPS with heterogeneous processing units, such that the schedules are resource-bounded to meet the physical requirements? For example, tasks that require significant computation time in a self-driving car can delay reaction, decreasing available braking time. Heterogeneous computing systems — containing CPUs, GPUs, and other types of domain-specific accelerators — offer effective capabilities to reduce computation time or energy consumption and expand such operating conditions. However, doing so under physical requirements presents several challenges that existing scheduling solutions fail to address.

We propose the creation of a structured system, the Constrained Autonomous Workload Scheduler (CAuWS). This structured and system-agnostic approach determines scheduling decisions with direct relations to the environment and differs from current ad hoc approaches which either lack heterogeneity, system generality, or this consideration of the physical world. By using a representation language (AuWL), timed Petri nets, and mixed-integer linear programming, CAuWS offers novel capabilities to represent and schedule many types of CPS workloads, real-world constraints, and optimization criteria, creating optimal assignment of heterogeneous processing units to tasks. We demonstrate the utility of CAuWS with a drone simulation under multiple physical constraints. The autonomous computation for the drone is made up of commonly used workloads (*i.e.*, SLAM, and vision networks) and is run on a popular heterogeneous system-on-chip, NVIDIA Xavier AGX.

CCS CONCEPTS

• **Computer systems organization** → **System on a chip**; **Heterogeneous (hybrid) systems**; • **Theory of computation** → *Parallel computing models.*

KEYWORDS

Heterogeneous computing, cyber-physical systems, physical constraints, parallel computing, system-on-chip

ICS '24, June 04–07, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656625

ACM Reference Format:
Justin McGowen, Ismet Dagli, Neil T. Dantam, and Mehmet E. Belviranli. 2024. Scheduling for Cyber-Physical Systems with Heterogeneous Processing Units under Real-World Constraints. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3650200.3656625

1 INTRODUCTION

Embedded heterogeneous computing systems offer improved latency and energy use in Cyber-Physical Systems (CPS), but to do so under the physical requirements of a CPS requires a way to simultaneously consider both different processor types and physical requirements. CPS has two factors that fundamentally limit the ability to handle these requirements. The first factor is physical — *e.g.*, braking distance, or battery charge. The other limiting factor is computational — *i.e.*, moving and processing data to make a decision takes time and energy. Many recent CPS embed powerful heterogeneous system-on-chips (SoC), such as NVIDIA's Xavier and Orin [20] or Tesla's FSDj platform [49], which can improve these computational limits.

A key property of these SoCs is that they employ a variety of processing units (PUs): often a general-purpose CPU and various, specialized domain-specific accelerators (DSAs). Many autonomous systems use DSAs for critical computations, such as object detection or tracking, or other 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 with half the energy of the GPU, at the expense of a higher latency [12, 13, 38].

Autonomous systems place multiple requirements on resource use. Systems with time-critical components or strict constraints on energy or power place upper bounds on resource use for any PU (*i.e.*, CPUs or DSAs); if computation takes more time, energy, *etc.*, such a system might fail. For example, an aerial drone may fly at 50 mph, when — considering the true computation time — it only has enough time to react if flying at 40 mph. Such requirements may pose a variety of constraints and objectives to optimize, such as, minimizing power while maintaining safety, flying as fast as safely possible, or minimizing latency. In some scenarios, multiple requirements (*e.g.*, minimizing time and energy) may result in a multi-objective optimization problem [35]. Optimal schedules must not only satisfy the physical constraints through assignments that execute tasks on alternate PUs, but must also try to optimize these objectives. Finding the optimal task schedule under such trade-offs

is non-trivial. The scheduling problem is NP-Complete even for single CPU systems with objectives. This challenge is exacerbated on heterogeneous architectures due to the high number of DSAs with diverse characteristics.

This paper addresses the challenge of *accounting for physical constraints while scheduling computational workloads in heterogeneous architectures that embed different types of DSA*. In particular, we overcome three limitations of existing approaches:

- State-of-the-art computational workflow modeling techniques for heterogeneous architectures [5, 10, 33, 41, 43, 46, 48, 54] *do not address the relationship between the physical dynamics of autonomous systems and the parallel and diverse capabilities of heterogeneous hardware.*
- 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 [31], but *such abstractions are wholly insufficient to express the varying performance and energy characteristics of different types of PUs in heterogeneous architectures* and 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 [25, 29]. Additionally, these works represent computation as a single task, neglecting the heterogeneity necessary for optimal execution.

In general, *the literature lacks a generalized methodology to represent the relationship between physical and computational constraints and to derive schedules that optimize the desired objectives of CPS with heterogeneous processing units.*

In this study, we propose CAuWS, the Constraint-based Autonomous Workload Scheduling for CPS with embedded heterogeneous processing units, which enables a generalized solution to the heterogeneous (*i.e.*, multi-DSA) scheduling problem that accounts for the physical constraints. To achieve this, our approach takes advantage of a novel representative language (AuWL), timed Petri nets, and mixed-integer linear programming. CAuWS, whose high-level operation is illustrated in Figure 1, is able to assign operations from a wide variety of workloads to PUs on heterogeneous architectures, creating a static schedule for resource-constrained and time-critical systems[1]. These schedules are optimal with respect to the user-defined objectives (including time) and profiled executions, and the schedules maintain the constraints defined in AuWL, as computation can impact time, power, or other factors limited by the physical requirements. CAuWS generates constraints from a problem definition, and we leverage the efficiency of highly-engineered constraint solvers, *e.g.*, [6, 17], to both optimally solve this NP-Complete problem and offer extensibility to add further constraints.

This paper makes the following contributions:

- To generally bridge between diverse hardware specifications, computation scheduling constraints, and physical concerns, we propose a formal representation that consolidates physical constraints, heterogeneous computational resources, and latency. We propose a timed Petri nets based representation for data flow, parallelism, and resource consumption.

[1]For further discussion on why static scheduling is feasible vs. dynamic scheduling for the class of problems this work tackles, please refer to Section 5.

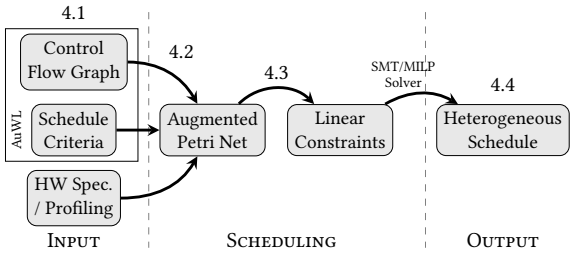

Figure 1: Overview of CAuWS.

- 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) [6, 17, 24] to find a globally optimal schedule.
- We define the Autonomous Workload Language (AuWL) as a front-end to our intermediate representation. AuWL specifies both data-flow and physical constraints, hence allowing rapid specification and high level consideration of a variety of autonomous problems. Other specification languages exist, but this specific application is novel compared to past languages, which either do not consider physical constraints or are limited to hardware-software co-design (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.

2 RELATED WORK

Scheduling for systems with heterogeneous PUs has been widely investigated over the last decade. A vast majority of key studies [3, 4, 8, 11, 32, 36, 53] are dynamic and typically use task-based heuristics. Static scheduling [1, 13, 23, 27, 45] is also common, with an emphasis on polyhedral code generation [2, 51] and genetic algorithms [15]. Our proposed approach differs significantly from these studies in its ability to map the physical dynamics of the system to the performance characteristics of heterogeneous computing.

A limited number of studies structurally approach timing in CPS computation by building models relating physical constraints and computational elements. For example, Krishnan *et al.* [29] create a computational model to map processing power to the weight of a CPS, and Wan *et al.* [52] 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. Alternatively, a few works have investigated integrating physical constraints into motion planning, such as balancing the energy between computation and movement [7, 47]. Most recently, Hadid *et al.* [25] 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 provide a generalized technique 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, 40, 55] and hybrid [19] systems. Relevant to this paper are works that establish timed Petri nets [22], use these to represent embedded systems [37], and utilize Petri nets for scheduling [40]. Studies [37] involving embedded systems used Petri nets for verification purposes. While Petri nets have been used for scheduling on single core CPUs [50], they have not been used for *scheduling* of diversely *heterogeneous* systems yet, to the best of 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 [26], robotics [14], and program verification and synthesis [18]. Specifically, modern and highly engineered constraint solvers offer a variety of techniques to efficiently address computationally hard problems [6, 17]. Previous scheduling techniques have also used constraint solving in an ad hoc manner. Additionally, many works using Petri nets have also applied constraint solving.

3 PETRI NETS FOR HETEROGENEOUS SCHEDULING

This section reviews key details of Petri nets and the extensions that 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 the parallelism of multi-processor 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.

[2]For thourough coverage of Petri nets, we refer the reader to texts such as [9].

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 still be non-negative, so transitions may only fire when places contain 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.

4 CAUWS: PROPOSED METHODOLOGY

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 by the operations and the data flow 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 runtimes 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 representation captures the parallelism and dependencies of tasks, data, and computational resources (*i.e.*, 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 [6, 17]) 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.

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

First, CPS workloads often 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, allowing static scheduling. Many tasks in CPS, such as the various computer vision networks, have fixed size inputs and no conditionals during 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 an object of interest and another mode to monitor that object. Other times the physical environment can change (with that change measured by sensors), and a CPS may want to adjust scheduling based on physical parameters such as

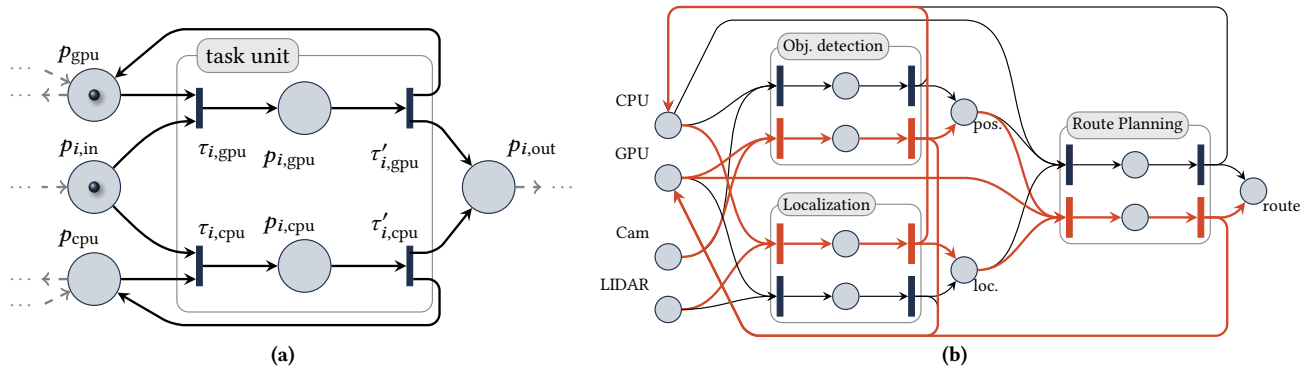

Figure 2: Proposed Petri net construction. (a) Petri net "unit" for one task, which is represented by the op identifier in the AuWL file. Each unit has multiple "paths", one for each PU. The unit operates by (1) firing $\tau_{i,xpu}$ to consume tokens for its input and a selected PU, and setting a token 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. (b) An example Petri net for a minimal control flow graph (CFG) with multiple PUs and tasks. Paths highlighted in red indicate a valid firing sequence. The highlighted firing shows the GPU running object detection task while the CPU simultaneously runs localization task. Then, the GPU runs route planning task.

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 Section 5.

4.1 System Specification

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, AuWL's 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.

Figure 3 contains an example AuWL representation for a simplified autonomous scenario, which must minimize execution time

```
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_box}
    op localization {in=lidar; out=localized_position}
    op route_planning {in=obj_bounding_boxes, localized_position;
      out=route}
}
```

Figure 3: An example AuWL program, defining the primary operations, their dependencies, CPS constraints and the objective for a simplified model of an autonomous vehicle.

under several constraints. The tasks are represented by the op keywords. `object_detection` and `localization` tasks process `camera` and `lidar` inputs and then output `obj_bounding_boxes` and `localized_ position` to the `route_planning` task. The keyword `constraint` 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`. 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.

4.2 Petri Net Intermediate Representation

We use the specification given in Section 4.1 to construct a Petri net intermediate representation, as shown in Figure 2. The Petri net captures the structure of control flow choices and resource use, facilitating the subsequent generation of constraints.

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. Data element places will correspond to either CPS inputs (*e.g.*, raw sensor readings) or CPS outputs (*e.g.*, actuator commands).

Then, we construct additional places, edges, and transitions according to the CFG defined in the AuWL specification. This construction consists of multiple units in the form of Figure 2a. For each PU to which a task i may be assigned, we create (1) a place ($p_{i,\text{xpu}}$) indicating the task i is running on PU xpu, (2) a transition $\tau_{i,\text{xpu}}$ indicating the task is starting on the PU, and (3) $\tau'_{i,\text{xpu}}$ indicating the task is finishing on the PU. We create edges that describe the availability of the PUs, the input, and the output. We create these units for every task in the AuWL specification.

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 constraints on shared resources, such as available power or energy. This capability is an advantage over typical CFG-only representations. 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 time-step, all CPS inputs are available, and at the final step, all CPS outputs must be available.

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

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

4.3 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 mapping between tasks and the heterogeneous PUs.

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 of inflow and outflow at each place,

$$p^{\langle k+1 \rangle} = p^{\langle k \rangle} + \overbrace{\sum_{j=1}^{|T|} {\tau_j}^{\langle k \rangle} W(p, \tau_j)}^{\text{inflow}} - \overbrace{\sum_{j=1}^{|T|} {\tau_j}^{\langle k \rangle} W(\tau_j, p)}^{\text{outflow}} \tag{2}$$

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 Section 4.2).

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.

4.4 Schedule Generation

Finally, we solve the constraints to obtain a schedule. We use a solver for Satisfiability Modulo Theories (SMT) — specifically, Z3 [6, 17] — to solve the constraints from Section 4.3. 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 variables ${\tau_{i,\text{xpu}}}^{\langle k \rangle}$ and all times $t^{\langle k \rangle}$ to determine the full heterogeneous schedule.

5 HANDLING DYNAMIC CONDITIONS IN CPS VIA STATIC SCHEDULING

Our proposed methodology in Section 4 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 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 via an adaptive set of static schedules.

5.1 Computation Time

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

- Input size: In a typical CPS, the input data is provided by fixed resolution devices, such as cameras or lidar. Moreover, many neural networks that CPS rely on are designed to operate on a fixed image or video frame sizes.
- CFG topology: While some CPS scenarios have branching control 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 object 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 NN every 10th frames and simpler tracking in-between [16]).
- 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.

	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

Table 1: CAuWS scheduling runtime (*i.e.*, overhead) in seconds for different size of task graphs. While these times may appear long, CAuWS supports pre-computation 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 [44], remains an area of future work.

On the other hand, some tasks, like the vision networks used in the case study in Section 7, require the same amount of computation regardless of the current input.

When *computation time* can be predicted statically, creating schedules before runtime 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.*

5.2 Physical Factors

Physical factors likewise present a challenge when creating static schedules. Cyber-physical systems operate in environments that have *changing* physical conditions (*e.g.* current distance to obstacles and 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 physical factors — *i.e.*, the switchover points where a change in 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 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 sub-optimal 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 precision takes prohibitively long. Instead, we use a recursive, binary partitioning of the n-dimensional hyperspace of physical parameters. Details are explained below.

5.2.1 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. Figure 4 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 an MILP. For example, the velocity constraints are nonlinear (v^2). However, if we take the 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 whether the vertices of a schedule region are the same. A positive answer to this check will guarantee that every schedule within the region is the same (see the following theorem). Such vertex checks are sufficient when all constraints are monotonic with respect to physical parameters, and thus reduce the number of checks we need to do.

Theorem 5.1. *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, then the entirety of R must share the same schedule. 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. (See Appendix A.3 for the proof.)*

This holds for our experiments as all constraints are monotonic with respect to obstacle distance, velocity, and temperature. The stopping distance strictly increases with velocity, power usage strictly increases with velocity, and maximum heat generation strictly decreases with ambient temperature.

5.2.2 Algorithm to Find Schedule Boundaries: To avoid the high number of CAuWS invocations that 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 1: Binary-space partitioning

We identify the scheduling regions using the binary space partitioning approach given in Listing 1. Based on Theorem 5.1, 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 that covers the space. An example of such partitioning is given in the upper-half of Figure 4.

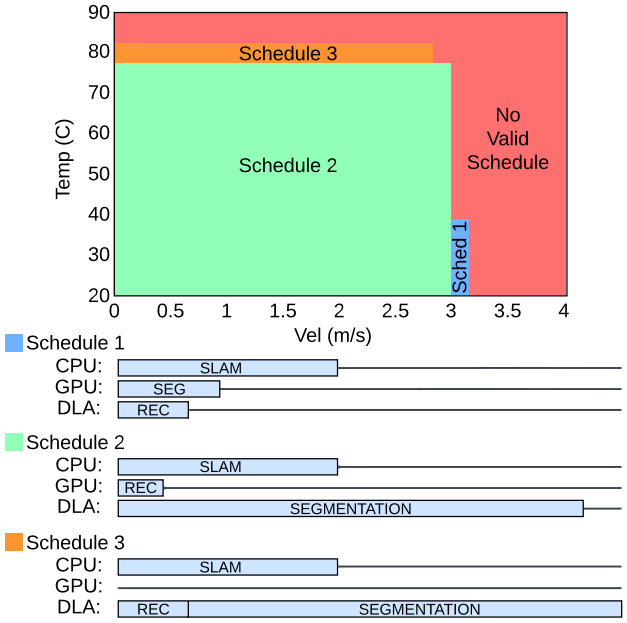

Figure 4: The set of schedules generated for the case study. CAuWS can pre-compute multiple schedules for varying physical parameters, creating a policy such as "if speed is less than 3m/s and ambient temperature is less than 76, run NN 1 on the GPU and NN 2 on the DLA simultaneously." This particular 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.

6 EXPERIMENTS

Our experiments include two simulation case studies and two sets of smaller tests. The two case studies involve a *simulated* Iris 3DR drone that must run various computer vision and planning workloads on an off-the-shelf SoC. Our case studies demonstrate the range of constraints CAuWS can consider at once. The first environment is a search-and-rescue task where we must also consider heat limitations due to a 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 separate.

While we consider drones in these case studies to demonstrate a more compelling example, CAuWS would work as well on any other CPS that meets the requirements for static scheduling (as previously explained in Section 5). 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 that CAuWS supports with a set of small synthetic experiments.

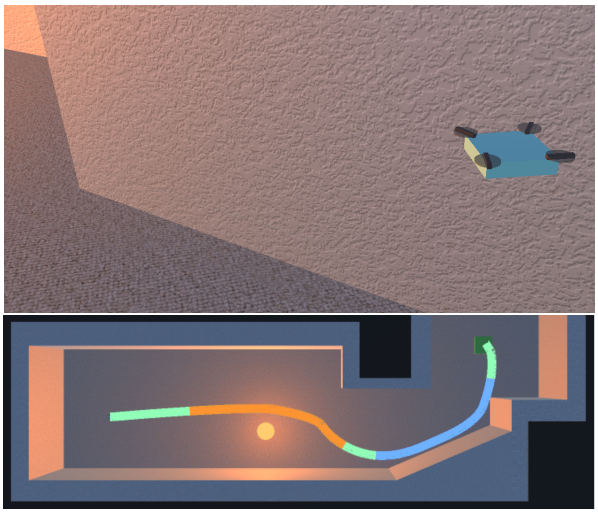

Figure 5: A screenshot of the simulation which includes quadcopter dynamics for a simulated 3DR-Iris quadcopter drone, positional heat sources (the orange dot), and obstacles. The highlighted path is colored based on what schedule CAuWS chooses, with different choices when close to the wall or heat source. These schedules correspond to those in Figure 4.

6.1 Methodology

The time, energy, and power use of the computational tasks must be predetermined for CAuWS. As the compute platform, we pick NVIDIA's Xavier [20] AGX and NX. They both provide two accelerators: GPU (low-latency) and DLA (low-power). We run and profile a variety of neural networks (NNs) supported by TensorRT [42] on both GPU and DLA. Data is 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 run these networks repeatedly and would not need warm-up. We profile ORB-SLAM [39] on the CPU with one to four reserved cores using the C++ chrono library. Profiling data is fed to CAuWS via a separate file listing latency, power, and energy consumption values.

7 CASE STUDY 1: ENVIRONMENT-LIMITED SEARCH AND RESCUE

In this section, we first explain our simulation environment and the corresponding constraints. 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.

7.1 Description

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 and built environments can also

disrupt radio communication, so autonomy is often necessary in 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 two DLAs.

The simulation follows a predefined route through a hallway corner with a heat source. Figure 5 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 of these tasks, it can plan its future actions on a solo CPU. We neglect this task because the time and energy it contributes is negligible, especially when run in parallel to the more intensive task. As CAuWS precomputes schedules before operation, the cost of selecting the appropriate schedule is also negligible.

7.2 Constraints

This environment induces a set of three constraints for CAuWS. Our novel DSL and Petri net based representation makes it trivial to support additional constraints for different scenarios. These constraints are as follows:

(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 at the given velocity and obstacle distance.

Additionally, when these constraints are satisfied and if there is some freedom left, we want to minimize latency and energy usage. The specific *linear* trade-off between energy and latency is defined in the AuWL file (Figure 6). While general multi-objective optimization can consider nonlinear trade-offs, MILP solvers can find the global optimum for linear objectives such as this.

The constraints are expressed to CAuWS through the AuWL file given in Figure 6. The system has a variety of ways to respond to the constraints. The scheduling targets include a CPU, GPU, and DLA to choose from. In general, for NN-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 usage. *These different responses create a set of three schedules (as depicted in Figure 4) that CAuWS chooses from over the course of the simulation.*

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 the autonomous loop body once.

The set of formal constraints these physical limitations creates is as follows. (See Appendix A.2 for a more detailed explanation of the algebra, assumptions, and determination of experimental constants.)

7.2.1 Heat: As heat is a constraining factor in computation, the fire in the simulation creates a rapidly varying temperature case which CAuWS must adapt. If ambient temperature is too hot, it must schedule tasks on the DLA instead of GPU to reduce the heat produced by the computation. 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 as total dissipated heat, E_{op} as the energy of each operation, and for an experimentally determined cooling rate k, the constraint can be represented as:

$$C \cdot \left(\sum E_{op} - k(T_{max} - T_{amb}) \cdot t^{\langle \text{end} \rangle} \right) \leq \Delta T_{sys} \leq 0 \qquad (3)$$

CAuWS can adapt different schedules as T_{amb} varies due to distance from the heat source.

7.2.2 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, acceleration and velocity.

Some constant power P_{aloft} must be spent to keep the drone aloft. Additional power is used to accelerate (P_{acc}) and maintain velocity (P_{vel}) against increasing air resistance. Although the drone will angle itself to produce the thrust for both hovering and horizontal forces at the same time, considering them separately will only slightly overestimate power and conservatively limit computation power. In this experiment, $P_{aloft} > 5(P_{vel} + P_{acc})$, so the effects are relatively minor.

$$P_{battery} > P_{accel} + P_{vel} + P_{aloft} + P_{comp} \qquad (4)$$

CAuWS selects schedules that obey this power constraint.

7.2.3 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, so that the system reacts in time and avoids unexpected collisions. In these situations, some tasks can be scheduled on GPU to reduce latency at the cost of energy. We can determine a maximum acceleration by assuming the drone directs all remaining power to its rotors. Using this value a_{max} we rewrite the stopping time as $\frac{v}{a_{max}}$, resulting in the following equation of motion:

$$D_{obst} > D_{stop} = v \cdot t^{\langle \text{end} \rangle} + \frac{1}{2a_{max}} \cdot v^2 \qquad (5)$$

D_{obst} varies over the simulation as the drone moves. CAuWS can adapt to this variation while satisfying other constraints in the meantime.

7.2.4 Constraint Linearization: MILP solvers support linear constraints; however, physical dynamics often contain non-linearities. For example, velocity v appears in the terms $vt^{\langle \text{end} \rangle}$ and v^2 above. Importantly, velocity is not a "decision variable" — *i.e.*, not an output of CAuWS. Following the algorithm we proposed in Section 5.2.2,

```
model case_study_one {
    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 (* batteryPow newtonsPerWatt)))
    cnstrnt (= stopDist (+ (* $svel $svel TIME )
      (/ (* $svel $svel $svel $svel) (* 2.0 maxAcc))))
    cnstrnt (< stopDist $distToObs)
    cnstrnt (= velPower (* velocityVsPowerConstant $svel))
    cnstrnt (> batteryPow (+ POWER velPower idlePower))
    cnstrnt (= tmpDelta (- maxTemp $ambTemp))
    cnstrnt (= maxTempOut (* TIME kCondctivity 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 6: The AuWL file corresponding to the first case study. It defines the control flow graph and places constraints on heat generation, power consumption, and latency.

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 Figure 4. Additionally, the steady state heat equation is used to avoid an exponential relationship between $\sum E_{op}$ and $t^{\langle \text{end} \rangle}$.

7.3 Results

The drone's travel results in ambient temperatures ranging from 27 to 73 $°C$, velocities up to 9.73$\frac{m}{s}$, and obstacle distances as low as 0.3m. A trace of this simulation, shown in Figure 5, records temperature, velocity, and obstacle distance. These values are fed into CAuWS, which then successfully finds the best schedules that satisfy the constraints and optimization objective. The accuracy of the physical conditions in the simulation relies on having accurate representations in the AuWL file. We determine the constraints in the AuWL file either from the 3DR Iris specification or experimentally from simulation, as previously described in Section 7.2. The resulting AuWL file is given in Figure 6.

7.3.1 Resulting Schedules: After profiling data is integrated, the solver outputs the set of schedules, as shown in Figure 4. As long as the drone remains within the region where there are valid schedules, CAuWS can find a schedule predicted to meet the constraints. The path taken is drawn in Figure 5, with changing schedules colored with different colors along the path.

The drone adapts to the varying conditions with these schedules. Along this path, the drone goes around the corner and skirts the wall, just before passing close to the fire. The corner reduces obstacle distance and required latency (schedule 1). As the drone approaches the heat source, a more energy-efficient schedule must be chosen (schedule 3). In other cases, it defaults to 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 path length. If the path went significantly closer to the wall or fire, there would be no possible schedules regardless of which schedule is used.

7.3.2 Predicted vs. Simulated Results: The calculated schedules are then run on the Xavier AGX as the drone follows 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. We identified that the misprediction originates from the shared memory contention as multiple PUs run concurrently and issue memory accesses. In our experiments, we also find that these errors of 8% and 14% (respectively) occur consistently in all iterations. The consistency easily allows a safety margin to be added to time constraints so that prediction errors due to shared memory contention [53] are mitigated. Such errors would likely be more permanently addressed by integrating slowdowns occurring due to shared resource contention into the Petri net based internal representation.

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

8 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. In this scenario, we use the NX version of Xavier series, which has a lower maximum power consumption when compared to the AGX version. These experiments demonstrate that CAuWS can handle different modes of operation by pre-computing schedules for dynamically changing environmental constraints.

8.1 Criteria 1: Power Limits

As explained in Section 7.2, the available power in a drone is shared between the computation and the actuation (rotors), and the total power a battery can provide is limited. Thus, rapid acceleration may require the NN to run on the more power-efficient DLA instead of the GPU. We test this scenario, which is simulated with the PX4 simulator [21], under the following objective and constraints in the AuWL input:

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

Figure 7 shows a timeline of the resulting power consumption. The adversary may have a higher velocity to which the drone must adapt, and this requires the drone to accelerate with higher power needs. While, in this scenario, computation uses only a fraction

of the physical power required, with a tight budget on the power, the schedule still must minimize the computation power. This is achieved by utilizing the energy-efficient.

8.2 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 by pre-computing the corresponding Pareto-optimal schedules, despite its static nature. When looking for an adversary, the drone must minimize power consumption to maximize its flight time. Once the drone detects and begins to follow the adversary, the scheduling requirements change. First, the drone now has a hard constraint on computation time to ensure that the adversary does not leave its field of view. Second, the drone must maximize the tracking accuracy (*i.e.*, trade-off between using efficient vs. accurate NNs), minimize its power use (*i.e.*, improving flight time), and minimize the total computation time. Assuming the drone can safely take longer to perform object detection via more accurate NNs, the quality of the object detection becomes most important. We represent this scenario using the following constraints and a linear combination of objectives in the input AuWL file:

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

Figure 8 shows the results of this multi-mode flight 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 while optimizing the other values in the priority order (accuracy, then power).

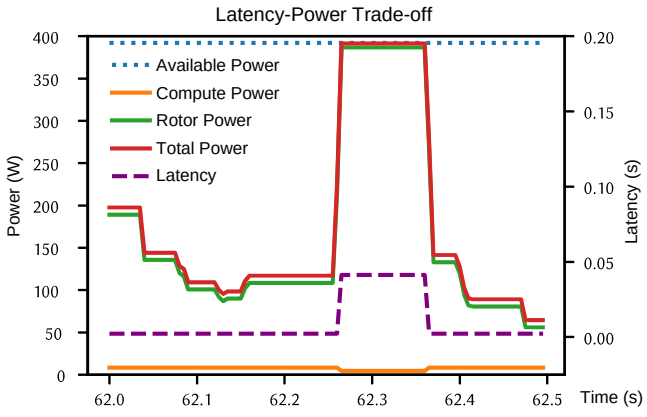

Figure 7: Power consumption in a pursuit flight. CAuWS balances power and latency under total power limits. Power values are maxed over a .05s window to demonstrate that CAuWS is able to rapidly adapt to changing power limits.

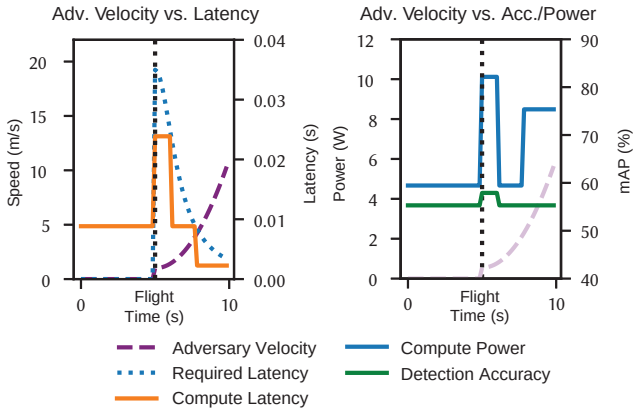

Figure 8: Search/Pursuit Modes Scenario: CAuWS can create schedules covering multiple operation "modes" in a simulated flight, enforcing different constraints for each. This example includes latency, network size, and power in a simulated pursuit scenario where the required computation speed increases when the adversary speeds up after detection.

9 COMPARISON WITH STATE-OF-THE-ART

To our knowledge, there are no related works that have 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
- Utilizing ready-to-execute task queues instead of considering the future tasks the CFG will follow

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 [28, 29] which, while it does not produce schedules, does select SoCs [29] and generated domain-specific SoCs [28] from a physically constrained Pareto frontier. The only factor, the stopping distance, that F-1 considers is also a constraint in our case study.
- Π-RT [34], an approach that can produce schedules for heterogeneous SoCs. It provides methods that attempt to either reduce latency or energy use. This technique, while it cannot consider arbitrary CFGs, can consider tasks at a similar granularity as CAuWS does. Unlike CAuWS, Π-RT cannot adhere to physical constraints, requires tuning of hyper-parameters, and cannot automatically adapt its scheduling approach to varying conditions.

9.1 F-1: Roofline Model for Physical Constraints

State-of-the-art constraint-based schedulers exist, but the constraints they target are limited to the ones that represent dependencies, latencies, *etc.*, and do not consider *physical* constraints. These schedulers rely on ad hoc techniques to place guarantees on latency or energy. The *F-1* technique [28, 29] is the closest comparison that considers *physical* constraints in relation to computational capabilities. However, F-1 does not make scheduling decisions and instead focuses on making HW design decisions. CAuWS is the first work that we are aware of, which combines both constraint scheduling

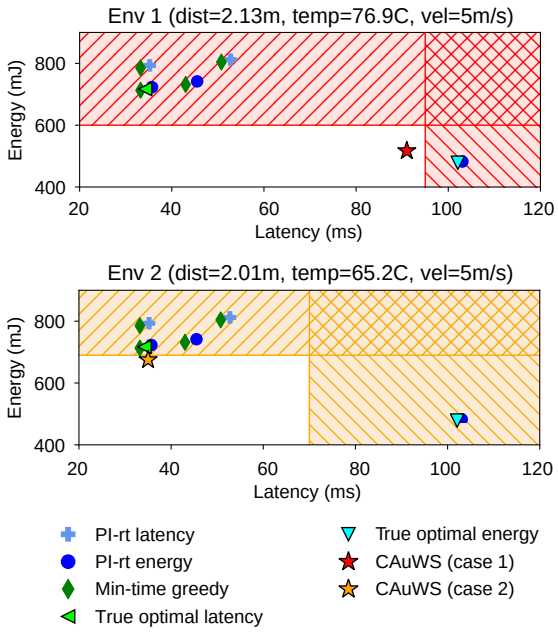

Figure 9: A comparison of the possible schedules that the two approaches of Π-RT [34], a time-first greedy algorithm and CAuWS generate. The two environments induce different time and energy constraints, which CAuWS can adjust for. All of the possible schedules from Π-RT and the greedy scheduler lie outside the constraints for these examples.

and physical considerations. F-1 proposes a mathematical approach to relate a single constraint, *i.e.*, the stopping distance limitation, to choosing an SoC that will be placed on a drone. The trade-offs that F-1 considers are between SoC weight, SoC latency, rotor weight, and rotor power. They produce extended *roofline models* to represent this relation analytically.

CAuWS differs by considering *scheduling* on *multiple* physical constraints. While F-1 only considers velocity, CAuWS considers constraints on heat, velocity, and power. This expansion is non-trivial as the optimum of the roofline models is a maximum overall constraint, while including multiple dimensions introduces trade-offs between values such that increasing one value could violate a constraint on another. CAuWS can also represent the relation in [29] with the following AuWL constraints:

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

where the value $\frac{1}{2a_{max}}$ (AMAXINV) divides the velocity into a valid and invalid region, similar to the one shown in Figure 4 - the border between these being the maximum velocity.

9.2 Π-RT: Scheduling for Heterogeneous SoCs

While many works that can schedule for heterogeneous SoCs exist, Π-RT [34] is the closest study, that we are aware of, to CAuWS. Π-RT and CAuWS share the same inputs (a set of tasks and profiling data) and outputs (a mapping between tasks and processors). There are three ways in which CAuWS demonstrates superiority against Π-RT, also as shown by the experimental comparison results given in Figure 9:

- CAuWS finds globally optimal solutions without tuning. Most studies on heterogeneous scheduling, such as Π-RT, often use heuristics to avoid the complexity of NP-complete scheduling. Although such work can handle much larger problems than CAuWS, their approaches result in nonoptimal schedules and often require fine-tuning of heuristic parameters. The need for supporting a large number of tasks can be unnecessary, as many CPS applications, especially those adhering to our assumptions, have relatively fewer tasks. As shown in Figure 9, CAuWS, without tuning, finds the most optimal schedule adhering to the given constraints. CAuWS is guaranteed by the MILP solver objective to find a Pareto-optimal schedule.
- CAuWS produces schedules that adhere to physical constraints while Π-RT does not consider physical constraints. Furthermore, CAuWS can find schedules that Π-RT cannot, some of which are necessary for the safe operation of the CPS. Shown in Figure 9 are all possible resulting 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 the schedules to these constraints. While Π-RT provides energy- and latency-first approaches, these have to be manually selected ahead of time based on prior knowledge of the tasks. CAuWS produces multiple schedules that can automatically adapt to these conditions and the result of this capability can be observed in Figure 9.

10 CONCLUSION

We present *CAuWS*, a system for representing and generating optimal schedules for CPS with heterogeneous PUs. Our approach could handle changing physical constraints for a variety of CPS scenarios. Its capability is demonstrated on a popular heterogeneous compute platform, NVIDIA's Xavier SoC, controlling a simulated drone. *CAuWS* offers a foundation to address an expanded set of scheduling problems that take into account physical constraints.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-2124010. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] Hamid Arabnejad and Jorge G Barbosa. 2013. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE transactions on parallel and distributed systems* 25, 3 (2013), 682–694.
[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.
[3] Mehmet E Belviranli, Laxmi N Bhuyan, and Rajiv Gupta. 2013. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–20.
[4] Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Laxmi N Bhuyan. 2018. Juggler: a dependence-aware task-based execution framework for GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 54–67.

[5] Arnamoy Bhattacharyya and Torsten Hoefler. 2014. Pemogen: Automatic adaptive performance modeling during program runtime. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 393–404.

[6] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *vz*-an optimizing SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015*. Springer, 194–199.

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

[8] Kevin J Brown, Arvind K Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A heterogeneous parallel framework for domain-specific languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 89–100.

[9] Christos G Cassandras and Stéphane Lafortune. 2008. *Introduction to discrete event systems*. Springer.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[11] Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. 2021. PiCAS: New design of priority-driven chain-aware scheduling for ROS2. In 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS).

[12] Ismet Dagli and Mehmet E. Belviranli. 2024. Shared Memory-contention-aware Concurrent DNN Execution for Diversely Heterogeneous System-on-Chips. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. 243–256.

[13] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E Belviranli. 2022. AxoNN: Energy-aware execution of neural network inference on multi-accelerator heterogeneous SoCs. In *DAC*.

[14] Neil T Dantam, Zachary K Kingston, Swarat Chaudhuri, and Lydia E Kavraki. 2018. An incremental constraint-based framework for task and motion planning. *The International Journal of Robotics Research* 37, 10 (2018), 1134–1151.

[15] Mohammad I Daoud and Nawwaf Kharma. 2011. A hybrid heuristic–genetic algorithm for task scheduling in heterogeneous processor networks. *J. Parallel and Distrib. Comput.* 71, 11 (2011), 1518–1531.

[16] Justin Davis and Mehmet E. Belviranli. 2024. Context-aware Multi-Model Object Detection for Diversely Heterogeneous Compute Systems. In *IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*.

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

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

[19] Isabel Demongodin and Nick T Koussoulas. 1998. Differential Petri nets: Representing continuous systems in a discrete-event world. *IEEE transactions on Automatic Control* 43, 4 (1998), 573–579.

[20] Michael Ditty, Ashish Karandikar, and David Reed. 2018. Nvidia's xavier soc. In *Hot chips: a symposium on high performance chips*.

[21] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. 2016. Rotors—a modular gazebo mav simulator framework. *Robot Operating System (ROS) The Complete Reference (Volume 1)* (2016), 595–625.

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

[23] Dominik Grewe and Michael FP O'Boyle. 2011. A static task partitioning approach for heterogeneous systems using OpenCL. In *Compiler Construction: 20th International Conference, CC 2011*. Springer, 286–305.

[24] Gurobi. 2022. Gurobi Optimizer Reference Manual. http://www.gurobi.com

[25] Ramyad Hadidi, Bahar Asgari, Sam Jijina, Adriana Amyette, Nima Shoghi, and Hyesoon Kim. 2021. Quantifying the design-space tradeoffs in autonomous drones. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 661–673.

[26] Henry A Kautz, Bart Selman, et al. 1992. Planning as Satisfiability.. In *ECAI*, Vol. 92. Citeseer, 359–363.

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

[28] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Sabrina Neuman, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. 2022. Automatic domain-specific soc design for autonomous unmanned aerial vehicles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 300–317.

[29] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. 2020. The sky is not the limit: A visual performance model for cyber-physical co-design in autonomous machines. *IEEE Computer Architecture Letters* 19, 1 (2020), 38–42.

[30] Sekhri Larbi and Slimane Mohamed. 2014. 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] Edward A Lee. 2015. The past, present and future of cyber-physical systems: A focus on models. *Sensors* 15, 3 (2015), 4837–4869.

[32] K. Li, X. Tang, and K. Li. 2014. Energy-Efficient Stochastic Task Scheduling on Heterogeneous Computing Systems. *IEEE Trans. on Parallel and Distributed Systems* 25, 11 (2014), 2867–2876. https://doi.org/10.1109/TPDS.2013.270

[33] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. The architectural implications of autonomous driving: Constraints and acceleration. In *ASPLOS'18*. 751–766.

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

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

[36] Xinxin Mei, Xiaowen Chu, Hai Liu, Yiu-Wing Leung, and Zongpeng Li. 2017. Energy efficient real-time task scheduling on CPU-GPU hybrid clusters. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[37] Ion Dan Mironescu and Lucian Vinţan. 2014. Coloured Petri Net modelling of task scheduling on a heterogeneous computational node. In *2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 323–330.

[38] Mohammad Alaul Haque Monil, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Allen D Malony. 2020. MEPHESTO: Modeling Energy-Performance in Heterogeneous SoCs and Their Trade-Offs. In *Proceedings of the ACM Intl. Conference on Parallel Architectures and Compilation Techniques*. 413–425.

[39] Raúl Mur-Artal and Juan D. Tardós. 2017. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Transactions on Robotics* 33, 5 (2017), 1255–1262. https://doi.org/10.1109/TRO.2017.2705103

[40] Martin Naedele. 1998. Petri net models for single processor real-time scheduling. *Citeseer* (1998).

[41] Dima Nikiforov, Shengjun Chris Dong, Chengyi Lux Zhang, Seah Kim, Borivoje Nikolic, and Yakun Sophia Shao. 2023. Rosé: A hardware-software co-simulation infrastructure enabling pre-silicon full-stack robotics soc evaluation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[42] NVIDIA. 2022. TensorRT. https://developer.nvidia.com/tensorrt

[43] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[44] Jussi Rintanen. 2012. Engineering Efficient Planners with SAT. In *ECAI 2012 : 20th European Conference on Artificial Intelligence*. 684–689.

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

[46] Kyle L Spafford and Jeffrey S Vetter. 2012. Aspen: A domain specific language for performance modeling. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

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

[48] Nathan R Tallent and Adolfy Hoisie. 2014. Palm: Easing the burden of analytical performance modeling. In *Proceedings of the 28th ACM international conference on Supercomputing*. 221–230.

[49] Tesla. 2021. Artificial Intelligence & Autopilot. https://www.tesla.com/AI. (Accessed on 11/20/2021).

[50] Jeffrey J. P. Tsai, S Jennhwa Yang, and Yao-Hsiung Chang. 1995. Timing constraint Petri nets and their application to schedulability analysis of real-time system specifications. *IEEE transactions on Software Engineering* 21, 1 (1995), 32–49.

[51] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* (2013), 1–23.

[52] Zishen Wan, Aqeel Anwar, Yu-Shun Hsiao, Tianyu Jia, Vijay Janapa Reddi, and Arijit Raychowdhury. 2021. Analyzing and improving fault tolerance of learning-based navigation systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 841–846.

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

[54] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. 2020. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1067–1081.

[55] Haitao Zhang and Feiyue Wang. 2005. A review of petri net based modeling and verification for embedded real-time systems. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Vol. 47411. 257–264.

A APPENDIX:

A.1 AuWL Grammar

The complete AuWL grammar in Backus-Naur Form (BNF) is given below.

⟨program⟩	→	"model"⟨NAME⟩"{"⟨program_body⟩"}"
⟨program_body⟩	→	⟨param_list⟩*⟨function⟩*⟨data_item⟩*⟨operation⟩*
⟨param_list⟩	→	"param"⟨params⟩
⟨params⟩	→	⟨param⟩
	|	⟨param⟩","⟨params⟩
⟨param⟩	→	⟨NAME⟩"="⟨expression⟩
⟨function⟩	→	"fun"⟨NAME⟩"("⟨name⟩")""="⟨expression⟩
⟨expression⟩	→	⟨term⟩⟨ADD_OP⟩⟨term⟩*
⟨term⟩	→	⟨factor⟩⟨MUL_OP⟩⟨factor⟩*
⟨factor⟩	→	⟨NAME⟩"("⟨expression⟩","⟨expression⟩*")"?
	|	⟨NUMBER⟩
	|	"("⟨expression⟩")"
⟨data_list⟩	→	⟨data_item⟩
	|	⟨data_item⟩⟨data_list⟩
⟨data_item⟩	→	"data"⟨NAME⟩⟨dimension⟩*⟨tags⟩?
⟨dimensions⟩	→	⟨dimension⟩
	|	⟨dimension⟩⟨dimensions⟩
⟨dimension⟩	→	"["⟨NAME⟩"]"
⟨operation_list⟩	→	⟨operation⟩
	|	⟨operation⟩⟨operation_list⟩
⟨operation⟩	→	"op"⟨NAME⟩⟨tags⟩
⟨tags⟩	→	"{"⟨tag_list⟩"}"
⟨tag_list⟩	→	⟨tag⟩
	|	⟨tag⟩";"⟨tag_list⟩
⟨tag⟩	→	⟨NAME⟩"="⟨values⟩
⟨values⟩	→	⟨value⟩
	|	⟨value⟩","⟨values⟩

A.2 Detailed Physical Constraints

A.2.1 Heat: To create a tractable linear constraint for heat, we model the steady state (temperature change is 0) heat flow instead of the current temperature. With T as temperature and total specific heat C, the equation for heat flow in the steady state is:

$$\frac{dT_{sys}}{dt} = C \cdot (P_{in} - P_{out}) \leq 0 \tag{6}$$

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

$$C \cdot \left(P_{comp} - k(T_{max} - T_{amb})\right) \leq \frac{dT_{sys}}{dt} \leq 0 \tag{7}$$

The dynamics of compressible fluid mechanics with heat sources is complex. Conservatively, we neglect additional convection from movement and air currents (which could only improve P_{out}) by experimentally finding k in a still room.

The maximum operating temperature is used for the steady state — if the flow at this point is 0, it can never exceed the maximum. Although this does overestimate the flow out, it conservatively guarantees that 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. The final constraint is:

$$C \cdot \left(\sum E_{op} - k(T_{max} - T_{amb}) \cdot t^{\langle \text{end} \rangle}\right) \leq \Delta T_{sys} \leq 0 \tag{8}$$

For the experiment, T_{max} is set to be the temperature of AGX reaching the rated $85°C$. To determine the other constants, an intensive program is run and an exponential function is fit to the resulting temperature curve. The die temperature and power were collected by reading the corresponding I2C addresses on the Xavier AGX. Knowing power consumption, this experiment determines the relationship between temperature increase and joules consumed (specific heat $C = 59.3J/°C$). It also determines the rate constant for cooling down ($k = 0.331W/°C$) based on the curve fitting.

A.2.2 Power limits: Power on a drone is shared between the rotors and the computation. The power output of a battery is also limited.

$$P_{battery} > P_{total} = P_{act} + P_{comp} \tag{9}$$

P_{act} can further be broken down into terms representing the required power to: (i) stay aloft, (ii) maintain a velocity against drag, and (iii) maintain thrust to accelerate. $P_{aloft} = 700.7W$ was determined experimentally by hovering in simulation.

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

$$P_{vel} = \sqrt{k_{air}/k_{thrust}} \cdot \sqrt{v} \tag{10}$$

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:

$$P_{accel} = k_{thrust} \cdot mass \cdot a \tag{11}$$

Combining these terms results in the equation:

$$P_{battery} > P_{accel} + P_{vel} + P_{aloft} + P_{comp} \tag{12}$$

In the experiment, $P_{battery} = 1800W$ was set to be the nominal maximum battery output from specification.

A.2.3 Stopping Distance: As discussed in Section 7.2, the drone has a "reaction time" due to latency. During this time $t^{\langle \text{end} \rangle}$ the drone will continue to move at its current velocity. Once this time is over, we assume that it can put the full power of the battery towards accelerating, represented in the term $a_{max} = k_{thrust} \cdot P_{battery} = 37.7622m/s^2$.

The basic constraint on the equation of motion is:

$$D_{obst} > D_{stop} = v \cdot t^{\langle \text{end} \rangle} + \frac{1}{2} a_{max} \cdot t_{stop}^2 \tag{13}$$

We can rewrite this equation to simplify $t_{stop} = \frac{v}{a_{max}}$ for the final equation:

$$D_{obst} > D_{stop} = v \cdot t^{\langle \text{end} \rangle} + \frac{1}{2a_{max}} \cdot v^2 \tag{14}$$

A.3 Theorem Proof

Proof for Theorem 5.1:

Let R be such a convex region.

Let the input variables be $x_1, x_2, ...$, be bounded correspondingly by $x_{1,min}, x_{1,max}, x_{2,min}...$

Let the unknown variables be $y_1, y_2, ...$, some of which (such as TIME) are the desired outputs.

Let all constraints be written as $c_1(x_1...) > 0, c_2(x_1...) > 0...$, and the objective be written as $O(x_1...)$. Assume there exist n bases $K = k_1, k_2...$ such that for any points X_1 and X_2 within the bounds, for all constraints, $K\dot{(}X_2 - X_1) \geq 0 \implies c(X_2) - c(X_1) \geq 0$. (This mathematically assumes that each constraint is monotonic, *i.e.*, strictly increasing or decreasing, even

if discontinuous, with respect to each variable. For example, in Sec. 7.2, for $vel > 0$ and $tmp > 0$, the following constraints

```
(< (+ (* vel vel) (* TIME temp)) 100)
```

would be allowed, but the following constraints

```
(< (* TIME POWER) 100)
(< (* (- vel 10) (- vel 10)) 100)
```

would not be allowed.)

For a proof by contradiction, let P be a point in R that does not share a schedule with the border of R.

As R is convex, for any basis direction we choose, there is a line L starting and ending on R that passes through P without intersecting R. We call the endpoints of this line Q_1 and Q_2. As the endpoints Q_1 and Q_2 of L are on the border of R, we know they share the same optimal schedule.

Starting at one endpoint Q_1, there are 2 cases where P could require a different schedule:

(1) An inequality constraint was violated by moving along L and keeping other values constant.
(2) The objective function resulted in a better optimum when moving along L, resulting in a different schedule being chosen.

For whichever case is violated by P, choose a line $K \cdot X$. By the earlier assumption, the $c()$ or $O()$ P violates is monotonic along this line, and, by having n as the basis, is reachable from the border.

For the endpoints Q_1 and Q_2 to share a schedule, but not P, this is a contradiction. In case (1), it would have to satisfy an inequality, then not satisfy it, then satisfy it again. This is not possible with a monotonic function. In case (2), the objective function is also monotonic. If P's schedule was chosen over Q_1, then it must be more optimal. As such, Q_2 must be at least as optimal as P, a contradiction. By contradiction, P must have the same schedule as Q_1 and Q_2. As P was an arbitrary point in R, any point inside R must share the same schedule.

It is possible to extrapolate to regions of any dimension through induction. The 0-dimensional case is a single point and trivially shares the same schedule. The regions of n dimensions can always be bounded by a combination of linear regions of dimension $n - 1$. For a target polytope (n-dimensional extension of polygons), the problem can be reduced to checking that its border $n - 1$ dimensional polytopes share a schedule, which reduces to checking their $n - 2$ borders, until we check 0-dimensional borders (the corner points).

Thus, any convex polytope with monotonic constraints bounded by points that share a schedule must entirely share a schedule.