VERIFIABLE COTS-BASED CYBER-PHYSICAL SYSTEMS

BY

STANLEY ZBIGNIEW BAK

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Associate Professor Marco Caccamo, Chair
Assistant Professor Sayan Mitra
Professor Lui Sha
Assistant Professor James Hill, Purdue University at Indianapolis
Abstract

Cyber-physical systems (CPS) use networked control software to interact with and manipulate the physical world. Examples of cyber-physical systems include smart buildings, power distribution networks, and fleets of autonomous agriculture vehicles. These types of systems are increasingly of interest due to the significant potential benefit of automating and optimizing tasks in the real-world and at large scales. However, before wide-scale deployment becomes a reality, two challenges must be addressed: safety and cost. The contained research directly addresses these two challenges, in the context of cyber-physical systems.

The second challenge of cyber-physical systems is their cost. Since cyber-physical systems interact with the physical world, these systems are often inherently real-time systems. In real-time systems, the correctness of a computation is not only a function of its result, but also depends on the timing at which the result is produced. For example, an inherently unstable airplane, like the F-16, needs a control system that can guarantee adjustments are always made dozens of times a second in order to guarantee aircraft stability. Most commercial off-the-shelf (COTS) computing systems, however, do not provide such real-time guarantees. Relying on custom-made components in order to guarantee timeliness properties, however, leads to systems with an exorbitant cost. For affordability, we must make use of low-cost COTS components. In the presented research, we address the primary problem with COTS components used in real-time systems: unpredictable interference, and therefore unpredictable timing, when accessing a shared memory resource. Methods are provided to mitigate both memory interference from external peripherals, as well as memory interference from other cores in a multi-core processor.

Since cyber-physical systems interact with the physical world, the effects of bugs in the design or implementation are not necessarily quarantined in the cyber (software) part of the system. Software written with traditional development practices will almost certainly contain bugs or unintended interactions among components. In CPS, these bugs can result in uncontrolled and possibly disastrous physical-world interactions. The safety problem for CPS is addressed on two fronts. First,
a technique based on selective command filtering is provided to give safety to the high-level CPS computation. This technique can guarantee distributed safety properties in the physical world, if assumptions are given about the low-level controllers. Second, a method for guaranteeing assumptions about the low-level controllers is presented. This method, based on the Simplex Architecture, allows safety invariants to be maintained in individual agents of the distributed CPS, despite the presence of bugs in their control software. Combined, the two approaches provide safety for entire CPS, without requiring complete formal verification of the system.
This dissertation and the work contained within would not be possible without help and support from a wide group of friends, colleagues, and family members.

I am first-off grateful to my committee members, professors, and research advisers who have guided me along the PhD process. I am thankful to have met my PhD adviser Marco Caccamo during recruitment week at UIUC, and his help in providing me with research direction ideas while giving me freedom to explore my research interests. Lui Sha helped guide me towards investigating Simplex in more detail and continuously provided insights and stories from his plethora of experience in real-time system research. Sayan Mitra helped drive much of the hybrid systems direction of the research and always inspired me with his technical insights. Ashley Greer and Jim Lenz from John Deere also helped advise much of the directions for the research, as well as its potential application.

When I started my PhD, I was fortunate to work closely with two Master’s students who were also starting off at the same time, Gbenga and Deepti, with which I took several classes, and who helped stay up late at night with me to get the System-Level Simplex demonstration working the night before the deadline. I also have fond memories of another all-nighter working on the Real-Time I/O Management System prototype with Rodolfo and Emiliano, where all three of us were debugging the system in three different ways, as well as planning CPS week in Chicago (and then working on a paper in the hotel room while we were there), and simply going out downtown to get our minds off research. I enjoyed discussing coding challenges with Gang, as well as visiting his apartment on weekend mornings after breakfast. Bach has also been a good friend during my PhD years, and I’ll especially remember getting up at 6:00 am to go to the gym with him in the middle of winter. I also value my friendship with Taylor, where we met randomly at a conference in California and ended up working together closely as well as training and running a half marathon. Sibin also served not only as a professional colleague relating to the security with real-time systems, but I also remember on occasion playing Texas Hold ’em with him into the night. I also am thankful to other friends I have met along the way which helped me during my PhD time at UIUC: Renato, Fardin,
Zhenqi, Karthik, Chrisy, Debessay, Kim, and surely many others.

A large thanks is deserved by my family, Mom, Dad, Aga and Kraig, Andy and Bethany, for always supporting me throughout the PhD process. Finally, thanks to my fiancee Xian for helping to distract me from working on research nonstop with things like teaching me Chinese and playing basketball, while at the same time, through example, inspiring me to work even harder.
Table of Contents

Chapter 1 Introduction .................................................. 1
  1.1 References and Acknowledgements .............................. 2

Chapter 2 Global CPS Safety ............................................. 4
  2.1 Safety using a Command Filter Safeguard ...................... 6
    2.1.1 Hybrid I/O Automata ....................................... 6
    2.1.2 System Definition ........................................... 7
    2.1.3 Safety Theorem .............................................. 7
    2.1.4 Application of Theorem to Runtime Checks ................. 10
  2.2 Guaranteeing Progress with Compatible Action Chains ........ 11
    2.2.1 Controller Architecture ................................... 11
    2.2.2 Compatibility and Stability ................................ 12
    2.2.3 Safety Guaranteed Run-time Checking ...................... 14
    2.2.4 Progress Property ......................................... 15
  2.3 Case Study: Coordinated Vehicle Flocking ...................... 16
    2.3.1 Desired Path Generation ................................. 19
    2.3.2 Intermediate Path Generation ............................. 19
    2.3.3 Implementation ........................................... 20
  2.4 Related Work .................................................... 21
  2.5 Conclusions and Future Work ..................................... 23

Chapter 3 Node Affordability using Commercial-Off-the-Shelf Components 25
  3.1 Real-Time I/O Management System .............................. 26
    3.1.1 Reservation Controller .................................... 27
    3.1.2 Real-Time Bridge .......................................... 30
    3.1.3 Evaluation ................................................. 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.3.3</td>
<td>Algorithm for Overapproximating $\text{Reach}$ and $\text{Reach}_\Delta$</td>
<td>105</td>
</tr>
<tr>
<td>4.2.3.4</td>
<td>Accuracy Convergence</td>
<td>110</td>
</tr>
<tr>
<td>4.2.3.5</td>
<td>Case Study: Waypoint Tracking System</td>
<td>112</td>
</tr>
<tr>
<td>4.3</td>
<td>Related Work</td>
<td>115</td>
</tr>
<tr>
<td>4.4</td>
<td>Conclusions</td>
<td>119</td>
</tr>
</tbody>
</table>

Chapter 5  Conclusion ................................................. 120

Bibliography ............................................................... 122
Chapter 1

Introduction

Cyber-physical systems combine communication, control, and computation in order to permit cyber components to directly affect the physical world while using a network for coordination. While this has the potential to create systems which efficiently perform precision operations in the real world, it also requires solving several challenges. Two recurring challenges in cyber-physical system design are the challenges of safety and affordability.

In earlier military and aviation applications, safety and reliability are achieved through sufficient redundancy and rigorous development and certification processes such as the DO-178B [1]. Such means and methodologies, however, are untenable for cyber-physical systems in other industries including automotive, agriculture, and off-road vehicles, because of a product cost difference of three orders of magnitude when compared with military and aviation applications. A low-cost approach to safety is required for CPS to become widespread in larger application spaces.

Large application spaces for cyber-physical systems demand a system design that is verifiably safe, and yet inexpensive. This important challenge is addressed directly in the contained research. This is done in three parts:

- Chapter 2 discusses how to provide safety at the global level. This method assumes local nodes have guaranteed behavior, but allows the network to be unreliable in terms of reordering or dropping packets.

- Chapter 3 provides methods to create low-cost real-time system by allowing the unverified majority of the system to be built out of commercial-off-the-shelf (COTS) components. The main challenge with COTS in cyber-physical systems is that they do not provide timing guarantees, so we create, implement, and evaluate hardware and software methods to overcome these intrinsic limitations.

- Chapter 4 addresses the issue of safety in the system in a formal manner. Based on prior research into the Simplex Architecture, we elaborate on an automated approach to generate
safety wrappers for untrusted components in order to reduce system verification cost. Such methods can provide the guarantees of local behavior necessary for global algorithm correctness.

The research presented here is ongoing, and a concluding look as well as future directions for research are outlined in Chapter 5.

1.1 References and Acknowledgements

The work presented in this dissertation is the culmination of six years of PhD research. I am extremely grateful to my PhD committee, Marco Caccamo, Sayan Mitra, Lui Sha, and James Hill, for their feedback in developing this work and driving my research direction. I am also thankful to my coauthors who have contributed to the presented work in various ways.

Some of the material presented here has previously appeared in academic proceedings [2, 3, 4, 5, 6, 7], and is copyrighted by the Institute of Electrical and Electronics Engineers (IEEE). Portions of this dissertation are reprinted, with permission, from:


Such permission does not in any way imply IEEE endorsement of any of the University of Illinois products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org, with the exception of ProQuest Information and Learning, which is permitted to supply single copies of the dissertation. By choosing to view this material, you agree to all provisions of the copyright laws protecting it.
Chapter 2

Global CPS Safety

As mentioned in the introduction, safety is a paramount concern for cyber-physical systems. In this section, we present techniques for addressing safety for a high-level algorithm controlling a cyber-physical system.

We consider a CPS scenario consisting of several embedded computing components each interacting and sensing the physical world and communicating with a central coordinator over an unreliable channel capable of dropping packets or causing unbounded message delay. Examples of such channels include wireless or the Internet. These embedded, low-level controllers attempt to accomplish some task in a coordinated fashion. Since the physical world is being manipulated, it is essential that the supervisory control logic is carefully designed and satisfies strict safety requirements. For example, autonomous vehicles may use wireless to communicate their positions and alter their future routes, but vehicles should never collide despite the potential for an unbounded number of message drops. This system is difficult to reason about because both (1) the communication layer can experience unbounded message delays and drops, and (2) the dynamics of the physical world are represented by interacting relationships in a continuous space. This complexity creates a significant potential for errors in both the design and implementation.

In this chapter, we aim to create a system capable of mitigating the effects of errors in the supervisory control logic of a cyber-physical system. The key enabler of our approach is the realization that, if the network is assumed to be unreliable, individual controllers must be able to maintain global safety even if packets do not arrive. In our approach, we propose a Command Filter Safeguard which is interposed between the supervisory control logic and the network, as shown in Figure 2.1. If supervisory control logic attempts to send control commands which may jeopardize system safety, the Command Filter Safeguard will selectively drop these packets, thereby maintaining the safety invariant. A key technical challenge with this approach is to determine the exact behavior of the Command Filter Safeguard for a particular system.

Since the network is unreliable, control commands which are sent may never arrive at the low-
level controller. For a system to be safe, therefore, it must account for this possibility of divergence in behavior, and satisfy a safety invariant both in the case where the command arrives and the new control strategy is used, and in the case where the command is lost and a previous control strategy remains in effect. We formalize a notion of \textit{compatible actions}, and then we show that the key to providing progress guarantees is to create finite \textit{compatible action chains} which end at a desired state. Since constructing these chains is the key to providing system progress, we provide an algorithmic method to aid in their construction.

This chapter is organized as follows:

- We formally define an unreliable communication channel in cyber-physical systems. We then prove that safety invariants can be preserved in systems with these channels by using run-time checks to perform selective packet drops. (Section 2.1)

- Selectively dropping packets can, of course, adversely affect system progress. We provide sufficient conditions which provide progress guarantees from a control-theoretic view of the CPS. This requires constructing compatible actions chains, as well as stronger guarantees from the communication network. (Section 2.2)
• We show how to apply both of these approaches in a coordinated vehicle flocking system. If the stronger assumptions hold, the system progresses towards detour points given by the operator at run-time. If not, the system still maintains a collision-free flock. (Section 2.3)

2.1 Safety using a Command Filter Safeguard

In this section, we use hybrid I/O automata to formalize the notion of a distributed networked control system with arbitrary delays and packetloss. We then prove a general theorem which is both a necessary and sufficient condition for showing safety of such systems. We then apply the theorem by stating the runtime checks in order to maintain system invariants, which will be encoded into the Command Filter Safeguard in the proposed architecture.

2.1.1 Hybrid I/O Automata

Hybrid input/output automata are general model for systems consisting of discrete and continuous states, where the discrete states are governed by transition rules, and the continuous states evolve according to differential equations. There is also input and output in these systems, which allows easy composition of different components into a larger system.

Rather than explaining the full semantics for hybrid I/O automata, we provide a brief overview of only the most important aspects here, and refer an interested reader to a more comprehensive review [8, 9].

A hybrid I/O automaton consists of four parts: variables, transitions, trajectories, and actions. **Variables** are the discrete or continuous entities of an automaton, for example velocity or mode. A state of an automaton is a specific valuation of the variables. **Transitions** provide the behavior of the discrete variables in the system. These have an enabling precondition and an effect. The state after the effect is applied is called the post state of the transition. Preconditions specify when transitions can occur, but generally automata are not forced to take a transition, which can create nondeterminism. **Trajectories** give the behavior of the continuous variables in the system as time passes, typically using differential equations, and systems can also have nondeterministic dynamics described by nondeterministic trajectories. The conditions under which time can not advance are given as stop conditions, which can be used to force an enabled transition to occur. Finally, **actions** indicate the interaction points for external communication with other automata. An action will always have a corresponding transition in the automaton. An action can occur when
both automata that have the action satisfy the corresponding transitions’ preconditions.

Time passes for a hybrid automata when a trajectory is acting upon the continuous variables. During the execution, there can be discrete jumps in state caused by the transitions. For two hybrid I/O automata with compatible actions, say $A$ and $B$, we denote their composition using $A \parallel B$.

2.1.2 System Definition

We model our supervisory control system as a network of communicating hybrid I/O automata. In this network, there is an automaton describing the behavior of each of the $N$ agents in the system, $A_1, A_2, \ldots, A_N$, and an automaton which models the communication channel. This model is slightly more general than the one discussed earlier with an explicit supervisory controller. Here, we could arbitrary choose one of the agents to be the supervisor.

In this section, we are concerned with checking for satisfaction of safety invariants for systems. We do this by providing a predicate on the states of the agent automata. We say the predicate is an invariant if it evaluates to true for all reachable states of the system from a given initial state. A system is a composition of the agent hybrid I/O automata and the communication automaton.

For our unreliable network, we consider a communication automaton with weak guarantees about message delivery, named $C_{\text{weak}}$, which can delay each message arbitrarily long, or drop it. Such an automaton matches the communication properties of many networked or wireless communication systems. The automaton description for $C_{\text{weak}}$ is given in Figure 2.2. Here, there are two possible send transitions, either of which can be applied when a message is sent out. The first one assigns a real-valued arrival time greater than the current time. The second one silently drops the packet.

We also will consider two other communication scenarios, $C_{\text{drop}}$ and $C_{\text{strong}}$. In $C_{\text{drop}}$, the first send transition of $C_{\text{weak}}$ is omitted so all messages get dropped. In $C_{\text{strong}}$, the second send transition is omitted, so that all messages can only be arbitrarily delayed, but never dropped. A communication automaton would be composed with each of the agent automata by connecting the receive transition with destination $i$ to Agent $A_i$. All the agents would invoke the same send transition.

2.1.3 Safety Theorem

In order to prove a predicate $P$ is an invariant for a system given a definition for each agent automaton and the communication automaton, a standard approach is to check that the invariant is satisfied for every transition and every trajectory. During this process, the invariant may need to strengthened in order for the proof to follow.

7
The standard approach for proving invariants, however, can be difficult to apply. Since reasoning is done ahead of time, the analysis must be applicable to all states which can be encountered for each rule.

The approach advocated in this paper is to use a combination of static reasoning done ahead of time along with runtime checks. With this approach, we can sometimes guarantee an invariant in an easier manner than by using the normal, static approach. Rather than reasoning over sets of possible values, we instead move part of the reasoning to runtime, and can therefore use a specific value in a specific message. In order to do this, however, we need a prove a theorem which provides an equivalent condition for verifying invariants.

We will now state the theorem formally and provide a proof outline. A system is described by a composition of the automaton for each of the agents \((A^N = A_1 \| A_2 \| \ldots \| A_N)\) and the automaton for the communication channel. A property \(P\) is the predicate we are trying to show is an invariant, and is a predicate on the states of the agents, \(P : A^N \rightarrow \{true, \ false\}\).

**Theorem.** A predicate \(P\) is an invariant for a system \(S = A^N \| C_{weak}\) if and only if (1) \(P\) is an invariant for the system \(S' = A^N \| C_{drop}\), and (2) from any post state of a receive transition in \(S\), \(P\) is preserved by the system \(A^N_{post} \| C_{drop}\), where \(A^N_{post}\) is the composed agent automata \(A^N\) starting in the post state of the receive transition.

**Proof outline.** First we show the direction that if conditions (1) and (2) hold, the invariant is satisfied by the original system.

The proof of this statement is based on the observation that at every point in time, either no messages have been received, or there is a most-recently received message by one of the agents. As
Figure 2.3: For every trace, at each time instant, either no message has been received in the system, or there is a most-recently received message.

shown in Figure 2.3, for every possible trace there will be some amount of time where no messages have been received by any of the agents in the system, followed by a intervals of time where there is a most-recently received message.

Our proof proceeds by contradiction. Assume $t_i$ is the first time at which $P$ is evaluates to false in $S$. If $t_i$ occurs before the first message is received, this means that $P$ would also evaluate to false in $S'$ at time $t_i$, since up to this point the behavior of $S$ and $S'$ is identical. This violates condition (1).

Therefore $t_i$ occurs at or after a message has been received and processed. Let $t_m$ be the time of the most-recently processed message before time $t_i$ (the time at which the receive transition was invoked in $C_{weak}$). We apply condition (2) of the theorem at time $t_m$ and take $A^N_{post}$ as the composed agent automata in the post state of the receive transition in $S$. Since in $S$, $P$ evaluates to false before any further messages are received after $t_m$, this would mean it also evaluates to false for the system with agent automata $A^N_{post}$ and a communication automaton which does not receive any messages. This is exactly the case checked by condition (2).

Next we show the other direction, that if a predicate $P$ is an invariant for $S$, conditions (1) and (2) will hold. Again, we proceed by contradiction.

Assume condition (1) does not hold but $P$ is an invariant of $S$. The behaviors of $C_{drop}$ can be exactly simulated by $C_{weak}$, which means that $P$ can not be an invariant for $S$.

Next, assume the second case that condition (2) does not hold but $P$ is an invariant for $S$. In the context of the false case of condition (2), let time $t_m$ be the time at which the receive transition is invoked. Now consider a communication automaton which produces an identical behavior as $S$ until $t_m$ and then no longer receives messages. This behavior can also be exactly simulated by $C_{weak}$ (by
taking the dropping send transition for messages which would originally have an arrival time after $t_m$), which means that $P$ can not be an invariant for $S$.

Since both cases yield contradictions, if an invariant is satisfied in the original system, conditions (1) and (2) must also hold.

The two conditions of the theorem are therefore both necessary and sufficient for proving an invariant is satisfied for a system with unreliable communication over all time.

### 2.1.4 Application of Theorem to Runtime Checks

From a static-time analysis perspective, the theorem does not gain us very much since condition (2) needs to be evaluated every time any message can be received, which is difficult to reason about. However, at runtime, condition (2) may be easier to verify. This is the approach advocated, to check condition (1) at system design time and condition (2) at runtime, which by the theorem will guarantees that $P$ is an invariant of the system.

One challenge of this approach is that the necessary runtime analysis needs to be automated in software, which is done in our architecture in the Command Filter Safeguard. Since there may be nondeterminism from the dynamics of the agents, and since in general this may involve an infinite-time reachability computation, this may easy or hard depending on the specific system.

Another challenge is to specify the action to take if the analysis for the specific message indicates condition (2) is not satisfied at runtime. The system can not be allowed to take action based on the message, since it may lead to a state which violates the invariant. In our proposed design, these messages are actively dropped by the sender (never sent out). This preserves condition (2) for the system (since no messages will be sent out unless (2) is satisfied) which guarantees that $P$ will continue to be an invariant for the system. Of course dropping messages can adversely affect system progress, but it will only be done to maintain safety (if the predicate captures a notion of safety). In Section 2.2, we present sufficient conditions to guarantee progress which require, among other things, a stronger communication automaton, where messages can be delayed arbitrarily but not dropped.

Since we are proposing to drop messages at send time, we need to reason about possible system states when the packet will be received (since condition (2) deals with the system state upon message reception, not sending). This also may be challenging because it involves reasoning about which messages may be sent out in the future before the arrival time of the message, and possible message reorderings. For example, in Figure 2.3, message $m_4$ arrives before message $m_3$. The runtime
analysis at the send time of message $m_3$ needs to take this possible reordering into account. Also, in an unrestricted system, these messages can be sent from and arrive at different agents (for example $m_3$ may be from Agent 1 to Agent 2, while $m_4$ is from Agent 3 to Agent 4).

For specific systems, however, this analysis may be simpler. For example, systems which maintain sequence numbers in messages and only take actions on the most-recent messages received, do not have to consider reorderings. Systems like the supervisory control system we are considering have a single entity which sends command messages, and therefore we do not need to reason about command messages exchanged between other agents.

One advantage of doing these checks at runtime, however, is that we are able to adapt systems where the desired predicate is not actually an invariant in an unmodified system. Therefore, this approach can be used to help design systems where we want a $P$ to be an invariant. This runtime check and selective message dropping is exactly the behavior encoded into the Command Filter Safeguard. In this way, the system will maintain $P$, even if the original supervisory control logic does not, due to bugs in the design or implementation.

### 2.2 Guaranteeing Progress with Compatible Action Chains

We will now describe a manner in which we can guarantee system progress without violating safety. We assume a more specific CPS model here where each agent is running a stable closed-loop controller.

First, we discuss the distributed control system architecture that we consider more specifically in Section 2.2.1. Section 2.2.2 defines the notion of compatible actions in the context of the distributed control system and proposes methods of checking compatibility. In Section 2.2.3, we then show scheme of coordinated control that guarantees safety according to our earlier result from Section 2.1. Finally, Section 2.2.4 proves progress of the system under a stronger assumption of the communication layer.

#### 2.2.1 Controller Architecture

As before, we consider a distributed control system consisting of a collection of $N$ agents with a central coordinator. We assume that each agent receives commands only from the central coordinator. Each Agent $A_i$ has a local controller and a variable set point $S_i$. The set point of Agent $A_i$ can be changed through communication with the central coordinator. In general, a set
point indicates a single or a sequence of (i) actions $A_i$ will take, or (ii) goal states $A_i$ moving towards. For simplicity, in this section we will assume $S_i$ is a single goal position of $A_i$. That is, the local controller of Agent $A_i$ drives the agent’s continuous variables to move towards the set point $S_i$. When agent $A_i$ reaches an $\epsilon$-ball around the set point (for some fixed $\epsilon$), agent $A_i$ will report its arrival to the central coordinator by sending a progress update message. The central coordinator will then, upon receiving arrival messages from all the agents, send each agent its next set point. An execution of Agent $A_i$ can therefore be viewed as a hybrid sequence 

$$\eta_i = wait_i[0] \leadsto receive[1] \leadsto \tau_i[1] \leadsto send \leadsto wait_i[1] \leadsto receive[2] \leadsto \tau_i[2] \leadsto send \leadsto wait_i[2] \ldots,$$

where (i) each $\tau_i[k]$ is a trajectory moving to a particular set point $S_i[k]$, (ii) $send$ is the Agent sending the progress update message, (iii) $wait_i[k]$ is a trajectory when waiting for next set point, where agent $A_i$ stays within the $\epsilon$-ball of $S_i[k]$, and (iv) $receive[k]$ is an action invoked by the central coordinator’s send action, during which the set point of agent $A_i$ is changed from $S_i[k-1]$ to $S_i[k]$. In each trajectory $\tau_i[k]$, the initial state and the final state of the trajectory are within $\epsilon$-balls of successive set points of $A_i$. A global set point is defined as a collection of the local set points for each of the $N$ agents, and is denoted as $S^N$.

We are interested in two properties of the system, safety and progress. Safety is defined in terms of a predicate $P_S$ and the progress property is defined in terms of a global set point $S^N_{\text{final}}$. The system’s goal is for each agent to reach within an $\epsilon$-ball around its set point in $S^N_{\text{final}}$, while keeping $P_S$ as an invariant.

### 2.2.2 Compatibility and Stability

As we have shown in Section 2.1, for safety the central coordinator needs to reason about future states of $A_i$, and will therefore issue set points according to what states $A_i$ can reach. Reasoning about future states of $A_i$ can be done using reachability analysis. We denote $Reach_i[k]$ as the set of reachable state of $A_i$ under trajectory $\tau_i[k]$. The reachable set of the global system (the composed behavior of all the agents) is denoted as $Reach^N$. For safety of the system, we need to verify that $Reach^N$ satisfies the safety predicate $P_S$. Recall that a trajectory $\tau_i[k]$ of $A_i$ depends on two set points, $S_i[k-1]$ and $S_i[k]$, of $A_i$. For a specific set point $S_i[k]$, we check whether $P_S$ remains true over the composed $Reach^N[k]$ by computing the reachable set of states for each of the other agents. This property of safety for a new global set point captures the a notion of compatible actions.

**Definition** $S^N[k]$ and $S^N[k+1]$ are said to be pairwise compatible actions if the global state $x^N \in Reach^N[k]$ always satisfies $P_S$ when every $A_i$ moves along a trajectory defined by $S_i[k]$ and
The notion of compatible actions can also be generalized to \(n\)-way compatible actions. That is, given \(n\) collections of set points, we can say they are \(n\)-way compatible if the global state always satisfies \(P_S\) when every agent moves along a trajectory defined by any pair of the set points. Due the extra requirements, however, it is generally easier to construct chains of pairwise compatible actions. For this paper we will use pairwise compatibility, and perhaps investigate applications of \(n\)-way compatible action chains in future research.

For compatible set point synthesis and compatibility verification, computing the reach set given the two global set points is crucial. We consider two ways of computing the reach set or its overapproximation: (1) direct computation using a hybrid system model checker, and (2) overapproximation which leverages the local controller’s stability guarantee. Direct computation is the preferred approach for checking for compatible actions, however this computation can often become intractable. We therefore elaborate on the second approach which provides an overapproximation of the reach set.

In many application of embedded systems, a controller is designed to be \textit{locally exponentially stable} with respect to a given set point.

\textit{Definition} A controller is said to be \textit{locally exponentially stable} with respect to a set point, if there exist a neighborhood of the set point such that any trajectories starting from any state in a neighborhood of the set point, eventually converge to the set point. In addition, the distance between the trajectory and the set point decays exponentially over time.

The neighborhood is also called the \textit{region of attraction}, which defines the maximum region from where the set point will be reached. Even though the distance between a trajectory and a set point is exponentially decaying, the set point may never be reached \textit{exactly} in a finite amount of time. However, any \(\epsilon\)-ball around the set point will be guaranteed to be reached in finite time. We next show how the stability property of the controller can help to compute an overapproximation of reachable set.

According to the \textit{Lyapunov inverse theorem} [10], if a controller of agent \(A_i\) is locally exponentially stable with respect to a set point \(S_i\), there exist a Lyapunov function \(V_i\) such that (i) \(V_i\) is continuous, (ii) \(V_i\) has value 0 only at the set point and is positive anywhere else, (iii) along any trajectory of agent \(i\) in the region of attraction, \(V_i\) is decreasing.

We define sublevel set of function \(V\) as \(L_c(V) = \{x \in \text{dom}(V) | V(x) \leq c\}\), which is a subset of the domain of function \(V\) over which the value of the function is bounded by a constant \(c\). By (iii), along
any trajectory $\tau_i[k]$, denote $x_0$ be the first state of this trajectory, the value of $V_i$ should not exceed $V_i(x_0)$. Thus, the future states should remain inside the sublevel set $L_{V_i(x_0)}(V_i)$ of the Lyapunov function $V_i$. Then we can use the sublevel set of Lyapunov function as an overapproximation of the reach set of $A_i$.

We denote $L_i[k]$ as the sublevel set contains trajectory $\tau_i[k]$. Recall that the sublevel set is determined by the function $V_i$ and value $V_i(x_0)$. Notice that (i) $V_i$ is determined by $S_i[k]$, and (ii) the initial state of $\tau_i[k]$ is determined by the previous set point $S_i[k-1]$. Therefore, a useful property of $L_i[k]$ is that it depends only on the set points being sent to $A_i$. From properties (i) and (ii) of Lyapunov functions, by choosing $S_i[k]$ close enough to $S_i[k-1]$, the set $L_i[k]$ can be made arbitrarily small. This property provides us with an important method to help design chains of compatible actions.

**Compatible Action Chain Algorithm.** One approach to creating a compatible action chain is as follows. Start with an action chain consisting of just the initial set point and final goal set point. Check if the two set points are compatible. If not, create an intermediate set point between the two. By the reasoning above, the computed reach set will become smaller. By recursively splitting pairs of set points, the reach sets can be made smaller and smaller, which increases the chance that the pair of global set points will satisfy the safety predicate $P_S$ and therefore be a pair-wise compatible action.

This method, however, may not always converge to find a compatible action chain in a finite number of steps. If it does converge, though, it solves the key problem to guaranteeing progress (constructing chains of pairwise compatible actions) in an automated manner.

### 2.2.3 Safety Guaranteed Run-time Checking

We assume low-level controllers which are locally exponentially stable, starting from a safe global set point.

We will now formally state the behavior of the supervisory control logic:

1. Until receiving progress report updates from all the agents, indicating that each agent is within an $\epsilon$-ball of the current way point $S^N[k]$, the central coordinator will not send any new set points.

2. The server computes a new set of set point $S^N[k + 1]$ following conditions below, and issues them to the corresponding agents.

   2a) The global set point $S^N[k + 1]$ should be compatible with the global set point $S^N[k]$. That
is, the reach set or its overapproximation does not violate the predicate $P_S$.

(2b) For each agent, the $\epsilon$-ball of its way point in $S_i[k]$ should be contained by the region of attraction of its way point in $S_i[k+1]$, to guarantee that the next set point will be reached by the low-level controller.

We now prove that safety predicate $P_S$ is an invariant of the system, using the theorem from Section 2.1.

(1) If all packets get dropped at the beginning, the way points never change and $P_S$ remains true.

(2) Suppose that after a packet get delivered, all follow-up packets get dropped. The server will stop sending new set points since not all reports are received. No agent will further update its set point since the coordinator will not send any new set points. Agent $i$’s states will be remain in the pair-wise compatible reach set, for the current way point, forever. By pair-wise compatibility, $P_S$ will remain true.

By (1) and (2), we conclude that $P_S$ is an invariant of the system.

### 2.2.4 Progress Property

We will now discuss a sufficient condition to guarantee system progress. Formally, we want the system reach a target global set point $S_{f_{final}}$ in some finite amount of time.

To guarantee progress, we require three assumptions. First, messages in the network can only
get delayed arbitrarily long, but cannot be dropped. For this assumption we will use automaton $C_{\text{strong}}$, as described in Section 2.1.2. In practice, this can be done by having a low-level network layer which keeps resending packets until an acknowledgement is received, assuming the connection will eventually get reestablished. Second, there is a finite chain of pairwise compatible actions from the current state to the target global set point $S^N_{\text{final}}$. Third, the local controllers for each agent are exponentially stable for each set point in the compatible action chain.

We will now prove that the system $A^N||C_{\text{strong}}$ meets our progress requirement. Recall that agent $A_i$'s execution is a hybrid trace $\eta_i = \text{wait}_i[0] \looparrowright \text{receive}[1] \looparrowright \tau_i[1] \looparrowright \text{send} \looparrowright \text{wait}_i[1] \looparrowright \text{receive}[2] \looparrowright \tau_i[2] \looparrowright \text{send} \looparrowright \text{wait}_i[2] \ldots$. First, $\tau_i[k]$ is a trajectory starting from an $\epsilon$-ball of $S_i[k-1]$ to an $\epsilon$-ball of the $S_i[k]$. Since we assumed the local controller is exponentially stable, the distance between the continuous state of $A_i$ and the set point is exponentially decaying. Thus, any $\epsilon$-ball of the set point will be reached in a finite time. Second, a send action through $C_{\text{strong}}$ takes finite delivery time to invoke a receive action of the coordinator. Since this is true for all agents, the coordinator will receive all the reports of progress in a finite time. At this point the next set point will be sent back to $A_i$. This sending also takes a finite time since it is done by $C_{\text{strong}}$. Due to this, the $\text{wait}_i[k]$ trajectory where $A_i$ is waiting for a new way point has a finite duration. Finally, since by assumption the chain of pair-wise compatible actions is finite, the target $S^N_{\text{final}}$ is reachable through finitely many of these steps. By this reasoning, we conclude that the execution of $\eta_i$ will reach $S^N_{\text{final}}$ in a finite amount of time.

2.3 Case Study: Coordinated Vehicle Flocking

In this section, we describe a vehicle coordination case study, where a single operator controls multiple vehicles over wireless. This is inspired by experimental off-road agriculture vehicle systems currently being investigated [11]. Many agriculture tasks, such as plowing, seeding, and harvesting, require a vehicle, or a fleet of vehicles, to perform a covering of the field. By using automation, the operating cost of such a system can be reduced since less people are required to run the equipment. Additionally, productivity and efficiency may also be improved since GPS-provided actuation may be more precise than what humans would achieve on their own, and since a large number of vehicles can be used at a time.

However, since vehicles need to use wireless in order to exchange control commands, care needs to be taken to show that this unreliable component can not cause, for example, collisions. This
imposes a challenge since, with unreliable communication, coming to a consensus among all the vehicles is generally not possible.

We consider the following system. A group of autonomous vehicles travel in formation along a path. The operator in the center vehicle can, at run-time, attempt to modify the flock's route by inputting a point into the supervisory control logic, called a detour point. The new, desired path takes the flock to the detour point, and then back again to the original path, maintaining the formation. Multiple detour points can be entered during operation. A simple case showing the potential danger in such a system is shown in Figure 2.5. Here, the detour point is shown as a red circle. If packets are lost, there are four potential locations where a collision can occur.

In the rest of this section, we will exploit the approaches described in the previous sections in order to create the supervisory control logic for such a system with the following guarantees:

- Vehicles do not collide with each other under packet loss or arbitrary packet delays.
- Despite packet losses or delays, all the vehicles end up in a pre-agreed location called $P_{final}$.
- The flock formation of the vehicles is maintained.

The central coordinator logic is physically on one of the vehicles which we call the leader. The leader is in charge of interacting with the operator, and generates control commands for each of the followers to be sent over wireless.

As described in Section 2.2, rather than immediately sending the final path to the followers, the leader will generate intermediate paths that are pairwise compatible and incrementally get closer to the desired path, as shown in Figure 2.6. In this way, wireless can be lost at any time and the system will remain safe, whereas if the wireless network works, the desired path will eventually be
Figure 2.6: A compatible action chain is shown for a pair of vehicles. Dotted paths are the new paths that are generated and solid paths are the ones whose progress update has already been sent. Notice that in each figure, whether the vehicle takes the existing path or the new path, collisions are avoided.

Every time new paths are generated by the leader, the Command Filter Safeguard will check the reachable region of each vehicle. Here, the reachable states for a vehicle are all the points that the vehicle can reach after it receives this new path. It therefore not only includes the new path of the vehicle, but also all the area in between the old path and new path that the vehicle might enter during the transition from the old path to new path. Once this region is calculated for each vehicle, if there is no intersection between the reachable regions, all the new paths can be sent out. If this check is false, we apply the method mentioned at the end of Section 2.2.2, where we generate a new intermediate path that is closer to the original path of the vehicles, and rerun the checks.

This method provides safety because no matter if the new path is received or dropped for any of the followers, the flock not collide because all the possible regions that the vehicle can reach were included in the reachable set and already checked for safety. At every step, the intermediate paths that are generated for the followers will be sent out only if all the progress update reports from all the other followers for all the previous paths have been received. As long as there is a vehicle that has not sent the progress update, the leader will keep sending the same path to the vehicle which has yet to report it is on the new path.

Next, we will forgo providing a formal automaton description of the leader and followers, and will instead give the technical details for the generation of a compatible action chain which can guarantee
progress while preserving safety. We first describe the generation of the desired path (which goes through the detour point), and then the computation of intermediate paths that form a pairwise compatible action chain to reach the desired path.

### 2.3.1 Desired Path Generation

Upon receiving a new detour point from operator, the leader will generate $\text{desiredPath}_i$, which is a path from the current position of the vehicle $i$ to the detour point and back from the detour point to the original path, which ends at the original endpoint $P_{\text{final}}$. For instance, the last path in the Figure 2.6 generated for the leader and follower is their desired path. In our implementation, we generate Bezier curves [12] to give a smooth path that transitions from the original path, touches the detour point, and then smoothly transitions back to the original path.

### 2.3.2 Intermediate Path Generation

After generating the desired path for each vehicle, the leader can not generally send this path directly to followers because, similar to the situation in Figure 2.5, the current paths and the desired paths are not compatible. Therefore, we use a method similar to the one described near the end of Section 2.2.2 to generate a chain of compatible intermediate paths, formally described below.

The path $\text{desiredPath}_i$ consists of $n$ segments with $n+1$ way points in which $\text{desiredPath}_i[0]$ is the current position of follower $f_i$, $\text{desiredPath}_i[n]$ is the original endpoint $P_{\text{final}}$, and one of the way points of this path equals to newly-entered detour point (or more strictly the point in the formation where follower $i$ should be when the leader reaches the detour point). When the leader is to send out a new path, each follower will be following its current path, denoted by $\text{currentPath}_i$ which also consists of $n$ segments and $n+1$ way points. Given this, a set of lines can connect each way point on the current path with the corresponding way point on the desired path. We define these lines as $L_i[0]$ to $L_i[n]$ such that for $0 \leq p \leq n$, $L_i[p].\text{startPoint} = \text{currentPath}_i[p]$ and $L_i[p].\text{endPoint} = \text{desiredPath}_i[p]$.

With these definitions, in order to generate the the $k$th way point on the $m$th intermediate path for follower $i$, called $\text{interPath}_i[m][k]$, we take the following incremental approach: $\text{interPath}_i[m+1][k] = L_i[k].\text{start} + \text{weight} \times (L_i[k].\text{end} - L_i[k].\text{start})$ where $\text{weight} \in [0, 1]$ is the size of incremental step. By making the weight closer to 0, we can make the intermediate path is closer to the original path. For maintenance of the formation, we could start with a lower $\text{weight}$ value to make sure the formation is not affected too severely if some of the vehicles receive the new path and some do not.
Figure 2.7: After new detour points are entered with the mouse (red dot), convergence of the flock can be observed in the simulator. The paths of each of the followers are shown in grey, the leader’s path is red, and the desired path of the leader is in orange.

Once $\text{interPath}_i[m+1]$ is calculated, we compute $\text{reach}_i[m]$, which is the reach set for follower $i$ upon receiving the path update. Alternatively, we could have used the Lyapunov function approach described in Section 2.2.2 which would need information about the low-level controller used by the followers. The reach set, assuming a controller which moves between way points exactly, is defined as the area between the current follower path and the potential new path (which includes all the transitions from the old paths to the new paths), bloated by the size of the vehicle. If $\forall i, j \in [0, n]$ and $i \neq j$: $\text{Reach}_i[m+1]$ is compatible with $\text{Reach}_j[m]$, then $\text{interPath}_i[m+1]$ can be sent to follower $i$ as the new path. Otherwise the incremental step is decremented by taking $\text{weight} := 0.9 \times \text{weight}$ and $\text{interPath}_i[m+1]$ is recalculated. This iterative recalculation will happen until an $\text{interPath}_i[m+1]$ with a compatible $\text{Reach}_i[m+1]$ is found, or a maximum number of trials is reached which indicates a chain of compatible actions could not be found.

The $\text{Reach}_i[m+1]$ sets that are computed after calculations of $\text{interPath}_i[m+1]$ are only valid if all the followers have already received $\text{interPath}_i[m]$. This is why the leader will only send $\text{interPath}_i[m+1]$ to the followers if a progress update message for $\text{interPath}_i[m]$ has already been received from each vehicle. If this is not true, the leader will not generate any new paths and will keep retransmitting the current paths until a progress update is received from all vehicles.

### 2.3.3 Implementation

We have implemented the described algorithm on the mobile robot simulator for the StarL platform [13]. StarL is a Java-based programming library for developing mobile robotics applications to control Roomba robots communicating over WiFi. It includes a simulator that runs identical robot logic code, but with simulated dynamics and network delays and drops.
The implementation consists of about 1300 lines of code on top of the base StarL platform, including both the logic for the leader and the followers. In the simulator, the user can use the mouse to click in the environment to enter new detour points. Since the StarL simulator is actually transmitting packets with delays, the progress of the compatible action chains can be seen in real-time. An example of this is shown in Figure 2.7.

We initially believed that the third progress property (a finite chain of pairwise compatible actions) would always be true. However, we discovered in the simulator that it could actually become false when the curve of the desired paths was sharper than the turning radius of the vehicle flock while in formation. If the another intermediate path was sent out, there would be potential for collisions. However, due to our checks, no incompatible intermediate path was sent out in this case, preserving safety. An example of this case is show in Figure 2.8.

We have prepared a video showing the execution of our algorithm on the StarL simulator [14].

2.4 Related Work

Networked control systems have been employed in a variety of industrial automation applications. Recently, industrial wireless protocols and products have been developed as replacements for wired control systems [15, 16]. These were made not only to reduce costs due to materials (wiring), installation and wire maintenance, but also provide benefits in flexibility by allowing easy modification to the existing communication infrastructure. One benefit of using these solutions is that they strive
to reduce (but can not eliminate) problems arising from communication delay and packetloss when wireless is used in industrial control systems.

Rather than sending control commands inside messages, a different approach to robustness with wireless control systems is to perform control computation using linear combinations of the states of neighboring nodes in a graph topology, where the neighbors will communicate using the unreliable wireless channels [17]. This was applied to multiple-input multiple-output systems and shown to be robust to limited failures of both communication channels and nodes. This approach provides graceful degradation of the control algorithm as the communication breaks down, but will not solve problems due to bugs in the high-level algorithm.

The Simplex Architecture [18] was developed as an approach to increase system safety for individual Linear Time Invariant (LTI) control systems, by filtering commands from an untrusted controller and switching over to a safe backup mode. This approach can compliment the one presented in this paper, by providing safety in the low-level controllers in a CPS architecture [19].

A network extension of Simplex has also recently been developed [20]. This work extended the Simplex approach to Linear Parameter Varying (LPV) systems, and incorporated network delays into the design. However, the analysis requires having a fixed upper bound on communication delay with no packetloss, which can not be guaranteed under wireless communication. Our guarantees of safety and progress hold without a fixed upper bound on communication delay, and, in the case of safety, we allow unrestricted packetloss to occur.

The described approach draws inspiration from the NASS framework developed to provide safety for medical systems communicating over wireless[21]. This system uses discrete dynamics with formal safety properties in a supervisory control system over wireless. Each command message includes a backup command vector, which is used if no further commands arrive. A safety filter provides protection from faults in the high-level control. This filter needs to reason about the worst-case packet delivery combinations, which in the case of the considered discrete system involves model-checking the possible combinations of packet reception and agent states. In our approach, we use a more control-oriented approach to providing safety and progress which allows for continuous state variables, and provide a method to help construct pair-wise compatible chains to guarantee progress.

Runtime approaches have been considered to create verified systems [22]. In this work, a time-bounded reachability computation is performed during system operation in order to determine if a controller should be disengaged. The advantage of this is approach is, since at runtime some of the variables are known, only a smaller state space needs to be considered. This is also the argument
we make when advocating the design of the Command Filter Safeguard.

For partially synchronous systems, where messages get bounded nondeterministic delays or dropped, a sufficient condition for verifying convergence properties has been established [23]. The sufficient conditions require that (i) messages get delivered infinitely often and (ii) there exist some invariant neighborhood topology of the system satisfying a Lyapunov-type property.

For asynchronous distributed systems, where messages get nondeterministic but bounded delay, a static approach for reasoning about the convergence of an asynchronous system has been proposed [24]. The approach shows that under some additional assumptions about the shape of the sublevel sets of the Lyapunov function, if convergence occurs in perfect communication, where messages get delivered instantly without dropping, convergence will also occur in the corresponding synchronous system.

2.5 Conclusions and Future Work

In this chapter, we have described an approach to increase resilience in a cyber-physical system from errors in the high-level control logic. Our approach, selectively dropping packets in order to maintain a safety invariant, is general and powerful, but comes at the cost of performing run-time checks to decide which packets need to be dropped. We have proven a theorem which states the exact condition that needs to be checked at run-time, in order to design the Command Filter Safeguard component.

Furthermore, we have used the notion of chains of pair-wise compatible actions to provide progress guarantees under a more reliable network layer. The challenge with this approach is the application-specific task of creating a finite chain of actions, which take the system from its current state to the goal state. We have provided an algorithm to help in constructing these chains.

As future work, we may investigate extending our progress mechanism from pair-wise compatible action chains to N-way compatible action chains, which N consecutive actions of a chain can be sent out without requiring the supervisory controller to wait for a progress confirmation message. The challenge with this direction is that it may be harder to construct an N-way compatible action chain, and it would need to be justified by an application which requires the extra flexibility.

We also believe our approach can be made applicable to hard real-time systems communicating over an unreliable network. Assuming there are synchronized clocks across the distributed system, we can include a timeout for each command message that is sent. If the command is received before
the timeout it will be acted upon, otherwise it will not. The guarantee provided would be that any command sent over the network will only be acted upon if it arrives within some time bound. This could be used, for example, in a traffic control system for intelligent intersections [25], where an autonomous car may be provided with a window of time during which it is allowed to enter the intersection. If the network delays this command too much, the car will not try to enter the intersection, and instead wait for a new command to be issued from the supervisory control logic.
Chapter 3

Node Affordability using Commercial-Off-the-Shelf Components

We now look further into the design of the individual nodes of the cyber-physical system. These nodes are computing systems which directly sense and actuate the physical world. In this sense, they are commonly embedded computing systems with real-time constraints. Real-time constraints mean that the computation done on the nodes needs to be not only correct, but also occur with guaranteed timeliness.

In this chapter, we will focus on the problem of affordability in the implementation of real-time computing systems. In order to reduce costs, we propose an approach which makes use of commercial off-the-shelf (COTS) components.

Commercial off-the-shelf computer software and hardware are often not only more affordable and perform than their custom-made counterparts, but also generally have better performance. For example, an inexpensive PCI Express bus [26] can transfer data three orders of magnitude faster than the expensive real-time SAFEbus [27]. Unfortunately, COTS components are typically designed paying little or no attention to worst-case timing guarantees required by real-time systems. Modern COTS-based embedded systems include multiple active components (such as CPU cores and I/O peripherals) that can independently initiate access to shared resources (like memory), which, in the worst case, cause contention which leads to a lack of timing predictability.

Computing precise bounds on timing delays due to this contention is difficult. Even though some existing approaches [28, 29] can produce safe upper bounds, they need to be very pessimistic due to the unpredictable behavior of arbiters of physically shared resources (like caches, memories, and buses). As a motivating example, we have previously shown that the computation time of a task can increase linearly with the number of suffered cache misses due to contention for access to main memory [30]. In a system with three active components, for example, a memory-intensive task’s worst-case computation time can nearly triple.

Here, we propose minimal additions to COTS systems and modifications to current practice, in order to make systems satisfy real-time computation requirements. The key idea for the approaches
in this chapter is that, in order take advantage the high average-case performance of COTS components without occasionally experiencing long delays being suffered by real-time tasks, we need to control the operating point of each shared resource and maintain it below saturation limits. Since COTS arbiters are typically work-conserving (meaning if there is a request pending they will grant it), avoiding saturation will avoid the non-real-time behavior of these arbiters.

Our implementations in this chapter, however, should be regarded as prototypes. For deployment applications, for example, one would not use FPGA development boards as we did. There is also some effort to be expected in order to apply the proposed approaches; they are not instant solutions which make off-the-shelf components behave in a real-time manner. The fundamental techniques that we investigate, however, are shown to be applicable to off-the-shelf hardware and software.

This chapter will elaborate on two approaches for making COTS systems more predictable, organized as follows:

• First, we will discuss the Real-Time I/O Management System [3, 31], which manages peripheral access to a shared memory in order to prevent contention both on the bus and at the memory controller (Section 3.1).

• Second, we will describe the PRedictable Execution Model (PREM), which can prevent contention between the CPU core and peripherals, as well as among CPU cores in a multicore system with a shared memory (Section 3.2).

3.1 Real-Time I/O Management System

Before describing the Real-Time I/O Management System, we first describe the way in which a COTS system typically works. A COTS system may include several commercial peripherals, such as video acquisition boards or network cards, plugged into standard buses, such as PCI or PCIe, on a commercial motherboard. Data from these boards travels through a series of bridges and buses (the specifics depend on the model of the motherboard), until it reaches main memory, where the CPU can read it through the Front Side Bus (FSB). Alternatively, the CPU could write data into main memory and instruct the COTS peripherals to retrieve it. For example, a network card could be instructed to upload packets which are stored in RAM.

Our proposed Real-Time I/O Management System, shown in context in Figure 3.1, adds two types of components to the existing COTS system. The first type is called the reservation controller, which implements the system-wide policy for accessing the bus. It can be thought of as a high-
Figure 3.1: The proposed Real-Time I/O Management System adds a reservation controller and real-time bridges to the COTS-based node.

level arbiter which instructs the real-time bridges to either communicate on the bus, or yield to other devices. The other type of component we introduce is called a real-time bridge, which is interposed between each peripheral and the communication bus. Each real-time bridge provides the actuation mechanism to enforce peripheral bus access. For rapid development, we implemented both of these components in hardware on field programmable gate arrays (FPGAs), although an industrial application would likely use an Application Specific Integrated Circuit (ASIC).

We describe the design of the reservation controller and the design of the real-time bridge, respectively, in Section 3.1.1 and Section 3.1.2. Then, in Section 3.1.3 we demonstrate the effectiveness of the approach with bus-level traces of the implementation in operation.

3.1.1 Reservation Controller

Multiple peripherals must cooperate to prevent a timeliness reduction caused by mutual interference while accessing a shared memory. The reservation controller centralizes decision making and coordinates multiple real-time bridges by instructing them to either forward or buffer peripheral traffic. Presently, we consider each peripheral as generating a single real-time I/O flow. If we were to allow multiple COTS peripherals to simultaneously access main memory we would have an effectively unpredictable bandwidth allocation (with a pessimistic bound), which is unacceptable for real-time computation. For this reason, we allow only a single real-time bridge to transmit at any one time. Therefore, we can consider the time allocated among all real-time bridges by the reservation controller as a shared resource akin to a monoprocessor CPU. In this analogy, each I/O flow is equivalent to a real-time task, and each I/O data chunk in the flow is equivalent to a job; transfer times for I/O data chunks are equated to computation times and can typically be derived
\begin{align*}
\text{block}_0 &:= \neg (\text{data}_\text{rdy}_0) \\
\text{block}_1 &:= \neg (\text{block}_0 \land \text{data}_\text{rdy}_1) \\
\text{block}_2 &:= \neg (\text{block}_0 \land \text{block}_1 \land \text{data}_\text{rdy}_2) \\
\text{block}_3 &:= \neg (\text{block}_0 \land \text{block}_1 \land \text{block}_2 \land \text{data}_\text{rdy}_3)
\end{align*}

Figure 3.2: These logical expressions, implemented in hardware on the reservation controller, provide a static-priority I/O scheduler for a four-peripheral system.

by dividing the I/O data amount by the achievable throughput of the real-time bridge.

Coordination between the reservation controller and each real-time bridge is achieved using two physical wires. Each real-time bridge communicates one boolean value, \(\text{data}_\text{rdy}\), to the reservation controller. This value indicates that data is buffered and ready to be sent on the bus. In turn, the reservation controller sends one boolean value back to the real-time bridge, \(\text{block}\), which instructs the bridge to either block I/O traffic or permit bus access. Real-time bridges instructed to block do not attempt to gain access to the bus, mandating the bus arbiter grants the unblocked peripherals access to the bus to send their data.

With only these two signals, many kinds of bus scheduling policies can be enforced. Consider, for example, scheduling four real-time bridges according to a static-priority bus scheduling scheme, such as rate monotonic (RM). Let \(\text{block}_i\) be the block command sent to the \(i\)th real-time bridge, and let \(\text{data}_\text{rdy}_i\) be the indicator of buffered data coming from the \(i\)th real-time bridge. Let the bridges be physically connected to the reservation controller in the order of their priorities (in the order of their rates for RM), from the highest priority bridge, \(i = 0\), to the lowest priority bridge, \(i = 3\). In order to provide static-priority scheduling on the I/O bus, the reservation controller hardware would implement the logical expressions in Code Block 3.2.

This setup, however, can also support a large class of monoprocessor scheduling algorithms which handle sporadic and aperiodic tasks using real-time servers [32]. In our prototype, for instance, we have implemented support for sporadic servers [33] under fixed-priority scheduling. Notice that the servers are implemented on the reservation controller and not on the associated real-time bridges. This decision has two major advantages. First, it removes the need for precise clock synchronization among real-time bridges and the reservation controller. Since all scheduling servers use the same physical clock, server budgets can be precisely calculated without clock skew. Second, it simplifies the interface between each real-time bridge and the reservation controller by reducing the number of physical wires to just two, \(\text{data}_\text{rdy}\) and \(\text{block}\). This becomes a concern for algorithms like EDF, where each server must communicate a precise deadline timestamp to the scheduling algorithm, which requires dozens of bits of information. By centralizing all scheduling servers on the reservation
entity sporadic_server is generic

PERIOD : integer := 1000; — microseconds
EXECUTION_TIME : integer := 500; — microseconds
CLOCK_FREQUENCY : integer := 100000000; — hertz
MINIMUM_EXECUTION_TIME : integer := 20; — microseconds
CEIL_OF_LOG_BASE_2_OF_PERIOD : integer := 10; — restore register data size
EXECUTION_TIME_DIVIDED_BY_MINIMUM_EXECUTION_TIME_PLUS_ONE : integer := 51;
CEIL_OF_LOG_BASE_2_OF_EXECUTION_TIME_DIVIDED_BY_MINIMUM_EXECUTION_TIME_PLUS_ONE :
  integer := 6 — determines number of restore registers (address size)
);
end sporadic_server;

Figure 3.3: The sporadic server VHDL code is parametrized based on the desired minimum execution time, which determines the amount of registers needed to store the server’s replenishment times.

controller, the number of physical wires is reduced, simplifying the electrical design.

One implementation consideration is on implementing a sporadic server algorithm in hardware. A sporadic server will consume the server’s budget when an aperiodic task is active. Then, after the server period elapses measured from the beginning of the task activation, the budget is replenished by the amount that was consumed. This means that the replenishment times and replenishment amounts must be stored somewhere. In addition, theoretically there are no restrictions on the amount of execution performed by the aperiodic task. For example, in one millisecond, an aperiodic task could be activated 10000 times and executes \( \epsilon \) time units each activation, requiring the storage of 10000 replenishment times during that millisecond. This was overcome in the implementation by enforcing a minimum execution time per peripheral, where access is granted to the bus for at least that amount of time, regardless of if the peripheral uses the entire time slice or not. The sporadic server in our implementation had a granularity of one microsecond, and VHDL generics were used to select the minimum execution time which determined the amount of memory that would need to be synthesized to store the replenishment times and amounts (as well as the server period and execution time). The VHDL entity description, which shows the generics of the component as well as the I/O interface is shown in Figure 3.3.
3.1.2 Real-Time Bridge

In order to provide real-time guarantees on bus communication, bus access must be controlled according to the policy dictated by the reservation controller. Since off-the-shelf peripherals are unlikely to have such a mechanism built-in, we interpose a device between each peripheral and the bus in order to provide this functionality to our real-time I/O management system. In addition to restricting bus access, real-time bridges also provide an important additional service to connected peripherals. Each real-time bridge provides a buffer which is able to store pending traffic while bus access is prohibited. This allows high-bandwidth peripherals to be blocked from the bus for relatively long periods of time without suffering from data loss due to full internal buffers on the COTS peripheral. Combined with a communication guarantee provided by the reservation controller’s scheduling policy, this guarantees the I/O system will deliver all communication traffic by its I/O deadline.

We envision a general real-time bridge using a setup similar to the ML455 [34], an FPGA development platform which can be directly interposed between a COTS bus and a COTS peripheral. As shown in Figure 3.4, this device contains both a PCI-X edge connector and a PCI-X socket slot connected to the same Virtex 4 FPGA chip, which would allow various types of peripherals to use the exact same real-time bridge if they have a PCI-X edge connector.

However, in order to rapidly develop a complete prototype, we focused our effort on a real-time bridge for one specific peripheral. We targeted a network interface card on the ML505 FPGA Evaluation Platform [35]. This device features both an Ethernet hardware interface, as well as a one-lane PCIe edge connector both connected to the same Virtex 5 FPGA chip. We now describe the hardware components of the System-on-Chip (SoC) design.

Figure 3.4: The ML455 can be interposed between the PCI-X bus and a PCI-X peripheral.
A logical outline of the important hardware components in the network interface card version of the real-time bridge is shown in Figure 3.5. We now describe each of these in the order of the dataflow through the real-time bridge during normal operation. Consider the case where a packet arrives through the Ethernet connection.

First, the physical hardware interacts with the Tri-state Ethernet MAC (TEMAC) hardware block. This is a fixed hardware block on the FPGA, and is the COTS peripheral that the real-time bridge is managing. This block maintains a set of memory addresses where to place packets after they are received. After the packet arrives to the TEMAC, it gets stored into the FPGA DRAM and an interrupt is raised to the Microblaze Soft CPU[36] which provides information on where the packet was stored and the size of the received data. The Microblaze processor is a soft CPU, meaning that is implemented using the reconfigurable logic on the FPGA. We developed a driver running on the Microblaze that will take the addresses and lengths of the packets and put them into a download queue of data to be sent to the main system. This queue exists in two parts. The potentially long tail of the queue is stored in software, whereas a bounded number of entries (say 128) of the front of the queue are stored in hardware on the Bridge DMA Engine. The Bridge DMA Engine, which is a hardware block we created specifically for the real-time bridge, manages actually moving the data out of FPGA DRAM and transferring it into the host CPU’s main memory. Along with the queue of data needing to be transferred, the Bridge DMA Engine manages the block and data_rdy signals on the real-time bridge. When block is asserted, the Bridge DMA Engine will not transfer data out of FPGA memory. The data_rdy signal is asserted whenever any data is in the
hardware queue. The DMA transfers themselves are abstracted as an address to address copy on the Processor Local Bus (PLB). The PLB / PCIe Bridge handles the process of translating a write transaction on the PLB bus to a write transaction on the PCIe bus. When the Bridge DMA Engine is unblocked by the reservation controller and performs a DMA operation, the memory containing the packet in FPGA DRAM is copied into the Host DRAM. After the transaction is complete, an interrupt is raised on the Host CPU, which then takes the packet data and passes it to the network stack for processing.

Sending packets out from the main CPU works in a similar way, but in the reverse order. The main CPU stores the packet data in Host DRAM and then writes the addresses to an upload queue which resides both in software as well as on the Bridge DMA Engine. When the Bridge DMA Engine is unblocked, it transfers the packets from Host RAM into FPGA DRAM (the PCI / PLB Bridge will again do address translation) and raises an interrupt to the Microblaze Soft Processor. Our driver on the Microblaze then sends the TEMAC hardware block the addresses and lengths of the packet data in FPGA DRAM. Finally, the TEMAC hardware sends the data over the physical Ethernet medium.

An important detail of this implementation is that the (comparatively) slow Microblaze processor does minimal processing (working with only the addresses and lengths) and no copying of the potentially high-bandwidth packet data. This allows our prototype implementation to achieve a network throughput of about 100 Mbps for upload and 80 Mbps for download, which coincides closely with Xilinx’s TEMAC performance benchmarks for our setup (ML505, 125 MHz Microblaze, 1500 byte Maximum Transmission Unit (MTU)) [37]. Performance could be further improved by using a larger MTU. Additionally, without the Microblaze bottleneck, the Bridge DMA Engine hardware was able to send data at 207 MBps, which approaches the theoretical limit of a single lane PCIe connection (250 MBps).

It is worth noticing that the proposed real-time I/O management system introduces additional latency compared with a COTS peripheral directly connected to the PCIe bus. This is one tradeoff that is made in order to provide control of peripheral bus access. We ran an experiment to get an idea of the effect of this additional latency by sending ping packets to the main CPU through our real-time I/O management system (with the real-time bridge scheduled as the highest-priority sporadic server) and sending ping packets directly to the FPGA’s PetaLinux OS. Surprisingly, the packet round-trip times through our bus scheduling prototype (2.40ms) were actually lower than the round-trip times for the packets processed immediately on the FPGA’s processor (2.62ms). Hence,
it is faster to place the packet data in the Bridge DMA Server, assert `data_rdy` to the reservation controller, wait for the `block` signal to be deasserted, receive access to the PCIe bus from the bus arbiter, transmit the data into Host DRAM, process the ping on the host’s 2.66 GHz CPU, and reverse the entire process for the ping response, than to handle the ping packet directly on the slower 125 MHz Microblaze processor. This experiment demonstrates the efficiency of our implementation.

### 3.1.3 Evaluation

There are two goals for our evaluation of the Real-Time I/O Management System. First, we show that there is a problem using COTS interconnect when a system has real-time requirements. We present an I/O task set which results in I/O deadline misses when run on a standard COTS bus. Next, we run the same I/O task set within our scheduling framework, and show that all deadlines are met. This demonstrates the non-real-time nature of COTS interconnect, and validates the correctness of our solution.

Performing direct measurements on a high performance COTS I/O system such as PCIe is difficult. The PCIe protocol implements point-to-point connections between each peripheral and the rest of the system running at the very high clock speed of 2.5 Ghz, making it hard to directly observe. In order to make the most accurate measurements, we instead used dedicated hardware on the reservation controller. Our trace acquisition hardware module polls the state of the `data_rdy` and `block` signals with a one microsecond resolution. Any changes in these signals, along with an associated timestamp, are output over the reservation controller ML505’s serial port where they can be received by an external computer for processing. Even this is not always fast enough, since events can be generated faster than they can be output over the serial port. If the send buffer in the debug hardware gets full, an error message is printed instead of sending misleading data. The internal buffers can also be reset over the serial port from the external computer, restarting communication of trace events.

We performed experiments on a COTS PC platform with an Intel 975X system controller (northbridge). The selected motherboard has four PCIe slots, allowing us to connect up to four high-speed peripherals. Using a PC platform permits easy access to all PCI slots, however, to derive meaningful measurements, we changed the FSB clock frequency obtaining a theoretical memory bandwidth of 2.4 Gbyte/s, which is in line with typical values for embedded platforms.

To make our experiments more easily repeatable, we instructed the smart bridge prototype to generate synthetic traffic instead of using traffic received by the TEMAC over the network. Our
<table>
<thead>
<tr>
<th>Board</th>
<th>Data Size</th>
<th>Transfer Time</th>
<th>Budget</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML555</td>
<td>4.0 MB</td>
<td>4.4 ms</td>
<td>5 ms</td>
<td>8 ms</td>
</tr>
<tr>
<td>ML505</td>
<td>1.1 MB</td>
<td>7.5 ms</td>
<td>9 ms</td>
<td>72 ms</td>
</tr>
<tr>
<td>ML505</td>
<td>1.1 MB</td>
<td>7.5 ms</td>
<td>9 ms</td>
<td>72 ms</td>
</tr>
<tr>
<td>ML505</td>
<td>1.1 MB</td>
<td>7.5 ms</td>
<td>9 ms</td>
<td>72 ms</td>
</tr>
</tbody>
</table>

Figure 3.6: Our experiments use four real-time flows.

Figure 3.7: This bus-level traces shows that I/O deadline misses can occur if all a standard COTS setup is used.

periodic task generating drivers run on the main CPU, and since our I/O schedule uses periods on the order of milliseconds, it is difficult to exactly synchronize all synthetic tasks. For this reason, we ran the tests for many hyperperiods, and show here the traces from the most closely aligned arrival times, which correspond to the *near-critical instants*. The arrival times of the presented traces are never separated by more than 0.8 milliseconds. Additionally, we implemented a traffic generator using an ML555 PCI Express Development Board [38] with a faster 8 lane PCIe connection. The synthetic traffic generator is programmed to send a constant amount of data to main memory every period and obeys the I/O scheduling commands from the reservation controller.

The task set used in our experiments consists of four real-time flows competing for main memory. The task parameters (data size, transfer time, period) are shown in Table 3.6. The task periods are harmonic, and the total utilization does not exceed 100%, so the task set is schedulable with the rate monotonic scheduling algorithm.

In the first experiment, the COTS bus is used without the reservation controller. Traffic gets sent on the bus as soon as it arrives, increasing the execution time of the ML555’s periodic task from 4.4 ms to over 8 ms (an increase of 82%) when the tasks start at a near-critical instant. This causes a deadline miss, as seen in Figure 3.7. In the second experiment, each peripheral is handled
Figure 3.8: Using the Real-Time I/O Management System, priorities are obeyed and all I/O tasks meet their deadlines.

by a sporadic server (whose corresponding budget and period are shown in Table 3.6) and all the servers are scheduled according to rate monotonic with total utilization $\frac{5}{8} + \frac{9}{72} + \frac{9}{72} + \frac{9}{72} = 1$. By using the Real-Time I/O Management System, the task set is now successfully scheduled without missing deadlines because the I/O traffic is prioritized. A trace of one hyperperiod starting at a near-critical instant is shown in Figure 3.8.

3.2 Predictable Execution Model (PREM)

In the previous section, we have shown how the Real-Time I/O management system can be used to enable system-wide scheduling of I/O peripherals. In addition to I/O peripherals, however, CPU cores will also access the same shared main memory resource. This sharing, and the associated possibility for contention, is a necessary result of the intended way high-bandwidth peripherals communicate with software: I/O peripherals directly write to main memory, and software running on a CPU core directly reads the data that was written in the memory. For CPU cores, however, we cannot use the same type of solution that we used for peripherals and buffer memory transactions. Not only is it likely infeasible to physically insert a buffering device between a COTS processor and motherboard, but also CPU cores stall while waiting for memory access, so buffering would negatively affect software computation time, and therefore real-time schedulability.

This section takes a different approach to provide high-level scheduling for CPU cores which we call the PRedictable Execution Model [39, 40]. PREM proposes modifying user-level applications in order to make explicit (and therefore schedulable) when programs can access main memory, and when they will perform computation on previously-cached data.
Figure 3.9: The modified I/O management system (compare with Figure 3.1) connects the peripheral scheduler to the PCIe bus to enable coordination with the CPU.

Under PREM, the task code is divided into a set of scheduling intervals, which are executed sequentially at run-time. The scheduling intervals are classified into compatible intervals and predictable intervals. Predictable intervals are further divided into two phases: a memory phase and an execution phase. During a memory phase, the CPU accesses main memory to fetch into last-level cache all the memory required for the rest of the predictable interval. Then, during the subsequent execution phase, the task performs computation on the previously-fetched data, without incurring any last-level cache misses. This allows a high-level scheduler to predict when the CPU core can access memory during predictable intervals, and when it will perform computation on already-cached data.

By using PREM, we can then enforce a high-level coscheduling mechanism among all the active components (both cores and I/O peripherals) in the system. This allows us to coschedule active components so that contention for accessing shared resources like memory is implicitly resolved by the high-level coscheduler component without relying on the low-level, non-real-time arbiters.

In particular, we will next discuss the PREM system and programming model (Section 3.2.1), and provide an evaluation of making real-time benchmark tasks PREM compliant (Section 3.2.2). After, we will discuss PREM scheduling for multicore systems by presenting a feasible algorithm to perform the scheduling which we have implemented and comparing the implemented multicore PREM algorithm versus other possibilities using a multicore PREM system simulator (Section 3.2.3).

### 3.2.1 PREM System and Programming Model

As in previous sections, we consider a typical COTS-based real-time embedded system composed of a CPU, main memory and multiple DMA peripherals. While we initially restrict the discussion to single-core systems with no hardware multithreading, we will later describe a PREM-based approach
for multicore systems. The CPU in the systems we consider can implement one or more cache levels. We focus on the last cache level, which typically employs a write-back policy. Whenever a task suffers a cache miss in the last level, the cache controller must access main memory to fetch the newly referenced cache line and possibly write-back a replaced cache line. Peripherals are connected to the system through COTS interconnects such as PCI or PCIe [26]. DMA peripherals can autonomously initiate data transfers on the interconnect. We assume that all data transfers target main memory, that is, data is always transferred between the peripheral’s internal buffers and main memory. Therefore, we can treat main memory as a single resource shared by all peripherals and by the cache controller.

**System Model.** The CPU executes a set of \( N \) real-time periodic tasks \( \Gamma = \{ \tau_1, \ldots, \tau_N \} \). Each task can use one or more peripherals to transfer input or output data to or from main memory. We model all peripheral activities as a set of \( M \) periodic I/O flows \( \Gamma_{I/O} = \{ \tau_1^{I/O}, \ldots, \tau_M^{I/O} \} \) with assigned timing reservations, and we want to schedule them in such a way that only one flow is transferred at a time. Unfortunately, COTS peripherals do not typically conform to the described model. As an example, consider a task receiving input data from a Network Interface Card (NIC). Delays in the network could easily cause a burst of packets to arrive at the NIC. Since a high-performance COTS NIC is designed to autonomously transfer incoming packets to main memory as soon as possible, the NIC could potentially require memory access for significantly longer than its expected periodic reservation. In Section 3.1, we discussed a solution to this problem through a Real-Time I/O Management System.

The architecture we will consider now, shown in Figure 3.9, is a modification on the Real-Time I/O Management System. We still include *real-time bridges*, interposed between each peripheral and the rest of the system. Each real-time bridge buffers all incoming traffic from the peripheral, and delivers it predictably to main memory according to a global I/O schedule. Outgoing traffic is also retrieved from main memory in a predictable fashion. To maximize responsiveness and avoid CPU overhead, the I/O schedule is computed by a separate *peripheral scheduler*, a hardware device based on the reservation controller from the Real-Time I/O Management System, which controls all real-time bridges. The peripheral scheduler in our architecture, unlike the reservation controller, is connected to the PCIe bus, and can therefore also coordinate with the CPU.

Notice that our previously-developed I/O management system does not solve the problem of...
memory interference between peripherals and CPU tasks. When a typical real-time task is executed on a COTS CPU, cache misses are unpredictable, making it difficult to avoid low-level contention for access to main memory. To overcome this issue, we now propose a set of compiler and OS techniques that enable us to predictably schedule all cache misses during a given portion of a task execution. The code for each task $\tau_i$ is divided into a set of $N_i$ scheduling intervals $\{s_{i,1}, \ldots, s_{i,N_i}\}$, which are executed sequentially at run-time. The timing requirements of $\tau_i$ can be expressed by a tuple $\{(e_{i,1}, \ldots, e_{i,N_i}), p_i, D_i\}$, where $p_i, D_i$ are the period and relative deadline of the task, with $D_i \leq p_i$, and $e_{i,j}$ is the maximum execution time of $s_{i,j}$, assuming that the interval runs in isolation with no memory interference. A job can only be preempted by a higher priority job at the end of a scheduling interval. This ensures that the cache content cannot be altered by the preempting job during the execution of an interval. We classify the scheduling intervals into compatible intervals and predictable intervals.

Compatible intervals are compiled and executed without any special provisions (they are backwards compatible). Cache misses can happen at any time during these intervals. The task code is allowed to perform OS system calls, but blocking calls must have bounded blocking time. Furthermore, the task can be preempted by interrupt handlers of associated peripherals. We assume that the maximum execution time $e_{i,j}$ for a compatible interval is computed based on static analysis techniques. However, to reduce the pessimism in the analysis, we prohibit peripheral traffic from being transmitted during a compatible interval. Ideally, there should be a small number of compatible intervals which are kept as short as possible.

Predictable intervals are specially compiled to execute according to the PREM model shown in Figure 3.10. They are divided into two different phases and exhibit three main properties. First,
during the initial **memory phase**, the CPU accesses main memory to perform a set of cache line fetches and replacements. At the end of the memory phase, all cache lines required during the predictable interval are available in last level cache. Second, during the following **execution phase**, the task performs useful computation without suffering any last-level cache misses. Predictable intervals do not contain any system calls and can not be preempted by interrupt handlers. Hence, the CPU does not perform any external main memory access during the execution phase. This property allows peripheral traffic to be scheduled during the execution phase of a predictable interval without causing any contention for access to main memory. Third, at run-time, we force the time length of a predictable interval to always be equal to $e_{i,j}$. Let $e_{i,j}^{\text{mem}}$ be the maximum time required to complete the memory phase and $e_{i,j}^{\text{exec}}$ to complete the execution phase. Then, offline, we set $e_{i,j} = e_{i,j}^{\text{mem}} + e_{i,j}^{\text{exec}}$ and at run-time, if the predictable interval completes in less than $e_{i,j}$, we busy-wait until $e_{i,j}$ time units have elapsed since the beginning of the interval. This property ensures that peripherals can transmit for at least $e_{i,j}^{\text{exec}}$ time units in a time window of length $e_{i,j}$. If we did not enforce a constant interval length, the execution phase could potentially complete in zero time, resulting in no peripheral traffic being sent during that predictable interval. Based on the constant interval length property, hard real-time guarantees can be provided to I/O flows [39].

Figure 3.11 shows a concrete example of a system-level predictable schedule for a task set with two tasks $\tau_1, \tau_2$ together with two I/O flows $\tau_{1/0}^{\text{I/O}}, \tau_{2/0}^{\text{I/O}}$ which service $\tau_1$ and $\tau_2$, respectively. Both tasks and I/O flows are scheduled according to fixed priority, with $\tau_1$ having higher priority than $\tau_2$ and $\tau_{1/0}^{\text{I/O}}$ higher priority than $\tau_{2/0}^{\text{I/O}}$. We set $D_i = p_i$ and assign to each I/O flow the same period and
deadline as its serviced task and a transmission time equal to 4 time units. As shown in Figure 3.11 for task \( \tau_1 \), this means that the input data for a given job is transmitted in the period before the job is executed, and the output data is transmitted in the period after. Task \( \tau_1 \) has a single predictable interval of length \( e_{1,2} = 4 \) while \( \tau_2 \) has two predictable intervals of lengths \( e_{2,2} = 4 \) and \( e_{2,3} = 3 \). The first and last interval of both \( \tau_1 \) and \( \tau_2 \) are special compatible intervals. These intervals are needed to execute the associated peripheral driver (including interrupt handlers) and set up the reception and transmission buffers in main memory (i.e. read and write system calls). I/O flows can be scheduled both during execution phases and while the CPU is idle. The described scheme can be modeled as a hierarchical scheduling system [41], where the CPU schedule of predictable intervals supplies available transmission time to I/O flows, and is therefore capable of providing hard real-time guarantees [39].

Executing a task according to the PREM model reduces its overall execution time because PREM ensures that peripheral traffic in main memory cannot contend with and stall cache fetches. In the worst case, the peripheral-induced delay can be very significant. A possible issue in our approach is that by deciding which cache lines to prefetch during the memory phase, we might need to prefetch more cache lines than the ones that are actually used at run-time in the execution phase. Our experimental evaluation in Section 3.2.2 shows that for several embedded benchmarks, this increase in memory load is not significant. A second possible issue is that by blocking I/O flows during compatible intervals, we risk reducing peripheral bandwidth significantly. However, in Section 3.2.2, we show that for a significant category of benchmarks, the execution time of predictable intervals dominates the total length of the job.

There are several necessary architectural considerations when using PREM in a system. We will first discuss cache eviction issues, and then write-back considerations.

Each time a cache line is prefetched with PREM, it has the potential to evict another cache line. This can cause two types of problems. First, the new cache line may replace some other cache line which was to be used in the upcoming execution phase (self-eviction). Second, it can evict a cache line previously prefetched by another task that has been preempted (storage interference). To prevent self-eviction, analysis of cache associativity and the cache replacement policy can be used to compute an upper bound on the allowed memory footprint of a predictable interval [39]. To avoid storage interference, several implementation solutions are available. In a single core PREM implementation, this problem is solved solely by limited preemption. In a multicore system, we could use an architecture where each core has an isolated cache. Another possible solution is to modify
the operating system to use cache coloring [42] to provide cache isolation among tasks [43, 44]. In a third solution, the last-level cache of each core can be partitioned directly among individual tasks in the system [45, 46].

It is also important to account for write-back in the last-level cache. During the execution phase, the task can modify cache lines in last-level cache. Since cache is partitioned, modified cache lines can only be evicted and written back to memory during further memory phases of the same task. There are two possible solutions to handle evictions. One approach, which we use in this paper, is to extend the worst-case execution time of memory phases to account for extra time used to perform write-back operations [39, 40]. In the worst-case, each fetch operation during a memory phase can elicit a write-back to occur. In practice, however, COTS hardware mechanisms (write queues) are typically very efficient at hiding the latency of write operations, which means this approach can be pessimistic. An alternative solution adds an explicit invalidation phase after each execution phase in a predictable interval. In the invalidation phase, write-backs can be forced [47] for modified cache lines that are not used in upcoming predictable intervals. In terms of scheduling, the invalidation phase can be scheduled as a memory phase. Hence, the same execution model and general scheduling results may be applied to both methods.

**Programming Model.** Our system supports existing applications written in standard high-level languages such as C. Unmodified code can be executed within one or more compatible intervals. We extend the source language with a **predictable** block construct that defines a single-entry, single-exit region of code that should execute as a single predictable interval. In C, we define the construct as the keyword **predictable** followed by a compound block of statements.

During compilation, the PREM real-time compiler transforms code within a predictable block so that it first prefetches any data and code required in the predictable block into the cache. Additionally, the compiler adds a busy-wait loop at the end of the predictable block to ensure that every execution of the predictable interval takes the same amount of time. This ensures that no cache misses occur during the execution phase and that the interval itself has a constant execution time.

In order to create a predictable interval, the programmer should first profile the code to determine the portions in which the task spends most of its execution time. Since it is difficult to compile arbitrary code so that it does not induce cache misses, there are several constraints that the compiler must place on code within predictable blocks. These constraints are:

1. Predictable code blocks should only access memory objects, arrays, and scalar values that are capable of being referenced at the entry of the predictable block. There should be no traversal
of link-based data structures (e.g., a binary tree) since the compiler cannot infer the memory that would need to be prefetched.

2. The code can use data structures, in particular arrays, that are allocated outside the predictable code block\(^2\). For global or heap allocated arrays, the programmer must specify the first and last address that is accessed within the predictable code block and (if necessary) the maximum difference between these two addresses if the compiler cannot infer this information via static analysis. This must be done for the code in the predictable block and for code within functions that are called (either directly or transitively) by the code in the predictable block. The compiler needs this information to add correct prefetching code to the predictable code block.

3. Code within a predictable block should not contain system calls, calls to heap allocators, or stack allocations within loops. System calls enter the kernel and execute code not generated by the PREM compiler, and the heap allocator executes code that can generate cache misses. Stack allocations cannot occur in loops because the compiler will insert code to prefetch the stack frame at the beginning of a predictable interval. Stack allocations in loops make the stack frame size difficult to predict.

4. Code within a predictable block should not make recursive function calls. Recursive function calls can grow the stack frame to an difficult-to-predict size, making it hard for the compiler to prefetch the stack frame.

5. Code within a predictable block may use both direct and indirect function calls. The compiler can use points-to analysis combined with call-graph construction [48] to find all the targets of indirect function calls. Since points-to analysis may yield conservative results, the compiler may find more function targets than are actually possible. If too many function targets are found (making construction of a predictable interval impossible), the compiler may ask the programmer to use annotations to specify the valid function targets.

The compiler employs several transforms to ensure that code within predictable blocks does not cause cache misses during the execution phase. First, the compiler inlines all functions called (either directly or transitively) by the predictable block into the predictable block (link-time optimization can inline functions across compilation units). This ensures that all code used by the predictable block is contiguous within virtual memory and uses a single stack frame. Second, the compiler inserts

\(^2\)Note that heap objects must have been allocated during a compatible interval.
code at the beginning of the predictable block to prefetch the code and data needed to execute the interval; this prefetching is done during the memory phase of the predictable interval. Based on the described constraints, this includes three types of contiguous memory regions: (1) the code for the function; (2) the actual parameters passed to the function and the stack frame (which contains local variables and register spill slots); and (3) the global and heap memory objects accessed within the predictable block. Third, the compiler inserts code to send scheduling messages to the peripheral scheduler as we will describe in the next paragraph. Finally, the compiler emits code at the end of predictable block to enforce its constant, predictable execution time.

In our model, peripherals are only allowed to transmit during a predictable interval’s execution phase or while the CPU is idle. To compute the peripheral schedule, the peripheral scheduler must thus know the status of the CPU schedule. Synchronization is achieved by connecting the peripheral scheduler to a peripheral interconnection as shown in Figure 3.11. Scheduling messages containing the amount of consecutive time in which peripherals are allowed to transmit are then sent by the task to the peripheral scheduler. In particular, at the end of each memory phase the task sends to the peripheral scheduler the remaining amount of time until the end of the current predictable interval. Note that to simplify the discussion we will not consider here issues of message propagation delay and clock drift in this paper, but the described scheme could be suitably modified by reducing the time allowed for peripheral traffic compared to the value contained in the scheduling message.

Finally, to avoid executing interrupt handlers during predictable intervals, a peripheral should only raise interrupts to the CPU during compatible intervals of its serviced task. As we describe in Section 3.2.2, in our I/O management scheme, peripherals raise interrupts through their assigned real-time bridge. Since the peripheral scheduler communicates with each real-time bridge, it is used to block interrupt propagation outside the desired compatible intervals. Scheduling messages are again used to notify the peripheral scheduler of the length of interrupt-enabled intervals. Note that blocking real-time bridge interrupts to the CPU will not cause any loss of input data because the real-time bridge is capable of independently acknowledging the peripheral and storing all incoming data in the bridge local buffer.

Converting existing code to predictable intervals clearly requires some amount of manual work. In particular, adding annotations to correctly split the code into predictable blocks requires some knowledge of cache parameters (since cache size is finite, whatever is prefetched must fit in memory). While this might seem an undue limit on code portability, the type of data-intensive, real-time appli-

---

3In our implementation we measured an upper bound to the message propagation time of 1µs, while we envision scheduling intervals with a length of 100-1000µs.
cations that we target in this work are typically already optimized based on hardware architecture. In this sense, we believe that the benefits of a more predictable behavior for program hot-spots, decoupled from the low-level details of bus and memory arbiters, outweigh the burden of code annotations. Additionally, we can design our compiler to help the programmer create predictable code blocks. The compiler can verify when the aforementioned restrictions are violated in a predictable block (e.g., it can use static analysis to find irregular data structure usage or use of system calls) and issue warnings to help the programmer to correct them. Furthermore, given cache size information, the compiler could verify that all prefetched memory regions fit in last level cache and issue warnings or errors otherwise.

A second possible concern is with regards to code constraints. In general, we believe that our constraints are not significantly more restrictive than those imposed by state-of-the-art static timing analysis. Typical hard real-time applications already avoid recursive calls, stack or heap allocation within loops, and indirect function calls that are not decidable at compile time. Furthermore, we are not aware of any timing analysis tool that can provide WCET bounds if Constraints 3 or 4 are violated.

Constraints 1-2 are more severe because they prevent using complex pointer-based data structures. However, existing code that is too complex to be compiled into predictable intervals can still be executed inside compatible intervals. An alternative solution [49] is to statically partition the cache, for example using the OS page allocator, into an area for predictable code and data, and a second area for complex, unpredictable data structures. During a predictable interval, the predictable area is prefetched while unpredictable data is handled by the caching logic. Static analysis can then be used to derive a (pessimistic) upper bound on the number of cache misses in the unpredictable area. Ideally, most of the data would be placed in the predictable area, resulting in a very small number of unpredictable misses. While we do not discuss it here, the PREM model could be amended to tolerate a small number of cache misses in the predictable phase by using analysis to compute the (limited) contention delay on both the task and I/O flows [29, 50].

3.2.2 PREM + I/O Evaluation

In order to verify the validity and practicality of PREM, we implemented the key components of the system. In this section, we first describe our implemented testbed. We then discuss our compiler implementation and analyze its effectiveness on several embedded benchmarks. Finally, using synthetic tasks, we measure the effectiveness of the PREM system as a function of cache stall
time.

3.2.2.1 PREM Hardware/Software Testbed

To support I/O flow scheduling, we developed a real-time bridge and peripheral scheduler prototype as described in Section 3.2.1. A full description of the implemented components is available [51], but here we instead only discuss the critical components. Compared to the Real-Time I/O Management System [52, 3], the new components exhibit two main differences: (1) there is an additional interrupt_block wire between the real-time bridge and peripheral scheduler is used to control interrupt propagation; (2) the peripheral scheduler is connected to the PCIe bus and exposes a set of registers accessible from the main CPU. In particular, there is a yield register which is used to receive scheduling messages, as described earlier. Both the real-time bridge and the peripheral scheduler require software drivers to be controlled from the main CPU and interact with each peripheral. The driver for the peripheral scheduler is extremely simple, exposing to the CPU the peripheral scheduler’s registers. The driver for each real-time bridge is more difficult, since each unique COTS peripheral requires a unique driver. As with the Real-Time I/O Management System, since we employ Linux (version 2.6.31), we can reuse existing, thoroughly tested Linux drivers to drastically reduce the driver creation effort [3]. The presence of a real-time bridge is not apparent in user space, and software programs using the peripherals require no modification.

For our experiments, we use an Intel Q6700 CPU with a 82975X system controller; we set the CPU frequency to 1Ghz obtaining a measured memory bandwidth of 1.8Gbytes/s to configure the system in line with typical values for embedded systems. We also disable the speculative CPU HW prefetcher since it negatively impacts the predictability of any real-time task. The Q6700 has four CPU cores and each pair of cores shares a common level 2 (last level) cache. Each cache is 16-associative with a total size of $B = 4$ Mbytes and a line size of $L = 64$ bytes; reloading the whole cache requires roughly 2.2 ms. Since we use a PC platform running a COTS Linux operating system, there are many potential sources of timing noise, such as interrupts, kernel threads, and other processes, which must be removed for our measurements to be meaningful. For this reason, in order to best emulate a typical uni-processor embedded real-time platform, we divided the 4 cores in two partitions. The system partition, running on the first pair of cores, receives all interrupts for non-critical devices (e.g., the keyboard) and runs all the system activities and non real-time processes (e.g., the shell we use to run the experiments). The real-time partition runs on the second pair of cores. One core in the real-time partition runs our real-time tasks together with the drivers.
for real-time bridges and the peripheral scheduler; the other core is turned off. Note that the cores of the system partition can still produce a small amount of unscheduled bus and main memory accesses or raise rare inter-processor interrupts (IPI) that cannot be easily prevented. However, in our experiments, we found these sources of noise to be negligible. Finally, to solve the issues with memory access due to paging, we used a large 4MB page size just for the real-time tasks using the HugeTLB feature of the Linux kernel for large page support.

3.2.2.2 Compiler Evaluation

We built a PREM real-time C compiler prototype using the LLVM Compiler Infrastructure [53]. LLVM is a mature compiler infrastructure with a front-end for many standard high-level languages including C, C++, and FORTRAN. We extended LLVM by writing self-contained analysis and transformation passes which were then loaded into the compiler. For simplicity, in the current compiler prototype, interval partitioning is performed by putting each predictable interval into a separate function. Within the predictable interval function, the programmer adds macros that 1) indicate that non-local data should be prefetched (PREFETCH_DATA(start_address, size)) and 2) indicate that the execution phase is beginning and send scheduling messages (START_EXECUTION(WCET)).

Our new LLVM compiler pass performs all remaining operations needed to transform the interval to be predictable. When a function representing a predictable interval is found, our transform first inlines all functions called within the predictable interval function. This ensures that there is only a single stack frame and segment of code that needs to be prefetched into the cache. Second, our transform inserts code to read and record the processor’s cycle counter at the beginning of the interval. Third, it inserts code to prefetch the stack frame and function arguments by prefetching memory between the stack pointer and slightly beyond the frame pointer (to include function arguments) using the x86 prefetcht2 instruction. Fourth, the transform prefetches the code of the function by deriving pointers to the beginning and end of the predictable function and then using the prefetcht2 instruction. Finally, the pass identifies all return instructions inside the predictable interval function and adds a special function epilogue before them. The epilogue performs interval-length enforcement by looping until the cycle counter reaches the worst-case cycle count based on the time value saved at the beginning of the interval and the predictable interval length (WCET) provided in START_EXECUTION.

To verify the correctness of the PREM compiler and to test its applicability, we used LLVM to compile several benchmarks mostly taken from MiBench [54], a commercially-representative embed-
The DES benchmark was modified to avoid memory access during the PREM execution phase by doing prefetching during the memory phase.

Table 3.1: The DES benchmark was modified to avoid memory access during the PREM execution phase by doing prefetching during the memory phase.

<table>
<thead>
<tr>
<th>Input bytes</th>
<th>4K</th>
<th>8K</th>
<th>32K</th>
<th>128K</th>
<th>512K</th>
<th>1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-PREM cache misses</td>
<td>151</td>
<td>277</td>
<td>1046</td>
<td>4144</td>
<td>16371</td>
<td>32698</td>
</tr>
<tr>
<td>PREM memory phase misses</td>
<td>255</td>
<td>353</td>
<td>1119</td>
<td>4185</td>
<td>16451</td>
<td>32834</td>
</tr>
<tr>
<td>PREM execution phase misses</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>104</td>
</tr>
</tbody>
</table>

As shown in Table 3.1, non-PREM execution results in a significant number of cache misses throughout the interval, which, as expected, increases roughly proportionally to the amount of processed data. If I/O peripherals were to transmit to main memory concurrently, the task’s execution time could increase. Conversely, the execution phase (after prefetch) of the predictable interval has almost zero cache misses, only suffering a small increase when a large amount of data is being processed. This demonstrates the key result: with PREM, I/O peripherals can communicate with main memory freely during the execution phase without affecting the timing of the executing task.

The reason the number of cache misses is not exactly zero in the PREM execution phase is that the Q6700 CPU core used in our experiments uses a cache replacement policy which can incur self-eviction, meaning that with more than one contiguous memory region, the probability of self-eviction is non-zero. CPUs which use different cache replacement policies could avoid this effect. In our experiments, we observed that the number of self-evictions is usually small. However, we still recommend that timing-critical applications avoid CPUs with cache replacement policy that may
cause self-evictions. Many embedded platforms used in safety-critical markets such as avionics use processors like the Freescale PowerPC family [55] with more predictable policies like pseudo-LRU [56].

A typical PREM code augmentation workflow was exemplified by the JPEG Image Encoding Benchmark. In this benchmark, we first used gprof to find that around 80% of the execution time is spent in the compress_data() function which performs DCT transformation, quantization and Huffman encoding.

We made compress_data() PREM-compliant by replacing constant function pointers with direct calls, adding 18 PREFETCH_DATA macros, and removing fwrite system calls from the predictable interval. The results for two image sizes are shown in Table 3.2, where time($\mu$s) represents the execution time of the whole interval.

We also went through the complete Automotive Program Group of MiBench and evaluated each of the six benchmarks to determine the broader necessity and feasibility of PREM compilation. Two of the benchmarks (basicmath and bitcount) were not data intensive, so PREM was not necessary (the effect of main memory contention on tasks which are not memory intensive is small). Of the remaining four benchmarks, three (qsort, susan_smooth, susan_edge) were found to be well-suited for PREM, and we were able to perform most of the computation inside predictable intervals. The final benchmark (susan_corner) had variable-size output, so PREM would typically need to prefetch much more buffer space than what was actually used. Notice that these types of tasks, however, where the amount of work done is not known ahead of time, would also perform poorly under worst-case timing analysis which is necessary for real-time schedulability analysis. The results are again shown in Table 3.2. Note that, except for susan_corner, the number of prefetched cache lines is only slightly higher than the number of cache misses suffered in the non-PREM way, indicating that the runtime overhead of PREM is generally low.

By evaluating several benchmarks, common strategies emerged to meet the PREM coding con-
contraints (discussed in Section 3.2.1). Similar to the JPEG benchmark, function pointers often pointed to a single function for a specific benchmark execution, and could therefore be replaced with a direct call (as with the qsort benchmark’s compare function). In the susan benchmarks, function calls such as malloc, memcpy, and memset were moved before the predictable interval function. Furthermore, since our current compiler implementation does not inline library functions, the implementation of selected libc functions was copied or rewritten to be local functions.

3.2.2.3 WCET Experiments

In this section, we evaluate the effects of PREM on the execution time of a task. To quickly explore different execution parameters, we developed two synthetic applications. In our linear_access application, each scheduling interval operates on a 256-kilobyte global data structure. Data is accessed sequentially, and we vary the amount of computation performed between memory references. The random_access application is similar, except that references inside the data structure are non-sequential. Both applications access all cache lines of the global data structure at run-time. For each application, we measured the execution time after compiling the program in two ways: into predictable intervals which prefetch the accessed memory, and into standard, compatible intervals. For each type of compilation, we ran the experiment in two ways, with and without I/O traffic transmitted by an 8-lane PCIe peripheral with a measured throughput of 1.2Gbytes/s. In the case of compatible intervals, we transmitted traffic during the entire interval to mirror the worst case according to the traditional execution model.

Figures 3.12a and 3.12b show the observed execution time for the scheduling interval as a function of the cache stall time of the application, averaged over 10 runs. The cache stall time represents the percentage of time required to fetch cache lines out of an entire compatible interval, assuming a fixed (best-case) fetch time based on the maximum measured main-memory throughput. Only a single line is shown for predictable intervals because experiments confirmed that, as predicted, injecting main memory traffic over the PCIe bus during the execution phase (where the task does not access main memory) does not increase execution time. In all cases, the computation time decreases with an increase in stall time. This effect is an artifact of our implementation: stall time is controlled by varying the amount of computation between memory references while the overall number of cache misses is kept constant. Hence, for lower stall time values, the task must execute for longer amounts of time. Furthermore, execution times should not be compared between the two figures because the two applications execute different code. Finally, note that reported execution
Figure 3.12: PREM decreases task execution time if the task originally has a random memory access pattern. For either pattern, PREM isolates execution time from peripheral traffic.

Execution times should be interpreted as average execution times rather than worst case measurements; each scheduling interval comprises thousands of cache misses, and experimentally capturing the worst case interference among all CPU cache fetches and PCIe peripheral transactions in main memory is extremely difficult.

In the random_access case, predictable intervals outperform compatible intervals (without peripheral traffic) by up to 28%, depending on the cache stall time. We believe this effect is primarily due to the behavior of DRAM main memory. Specifically, accesses to adjacent addresses can be served quicker in burst mode than accesses to random addresses. Thus, using PREM can actually decrease the total execution time by loading all the accessed memory into cache, in order, at the beginning of each predictable interval. Furthermore, note that transmitting peripheral traffic during a compatible interval can increase execution time by more than 60% in the worst case. In Figure 3.12b, predictable intervals perform worse than compatible intervals (without peripheral traffic). We believe this is mainly due to out-of-order execution in the Q6700 core. In compatible intervals, while the core performs a cache fetch, instructions in the pipeline that do not depend on the fetched data can continue to execute. When performing linear accesses, fetches require less time and this effect is magnified. Furthermore, the gain in execution time for the case with peripheral traffic is decreased: this occurs because bursting data on the memory bus reduces the amount of blocking time suffered.
by a task due to peripheral interference (this effect has been previously analyzed in detail [29]). In practice, we expect the effect of PREM on an application's execution time to be between the two figures, depending on the specific application's memory access pattern.

Additionally, we have also performed extensive experiments to validate the correctness of the scheduling message and interrupt blocking mechanisms, as well as obtained bus-level traces of PREM running on our testbed [51].

### 3.2.3 Multicore PREM

We will now discuss the application of the PRedicable Execution Model (PREM) for multicore system scheduling [5]. We will first briefly describe an approach which we have analyzed and implemented called Memory-Centric Scheduling. Although Memory-Centric Scheduling has a worst-case schedulability bound, it is not clear whether it is the best approach for scheduling a multicore system under PREM. This motivates the need for a larger evaluation of a different possible algorithms, which we perform using a custom-made PREM system simulator. We first describe an abstract task model for use in our simulator, and then present an evaluation of a number of algorithms. The best algorithm based on our evaluation, which we call M-LAX, schedules access to memory in a nonpreemptive fashion according to a global least-laxity-first policy.

#### 3.2.3.1 Memory-Centric Scheduling

An initial approach to scheduling PREM-compliant tasks in a multicore system is called Memory-Centric Scheduling [40]. Memory-Centric Scheduling provides core isolation by using a high-level Time Division Multiple Access (TDMA) schedule for granting access to main memory. The scheduling policy on each core promotes the priority of memory phases over execution phases when memory access is permitted by the high-level schedule. This is done to ensure that each memory phase completes as soon as possible, hence allowing the corresponding execution phase to take place between TDMA slots. For the same reason, memory phases are not allowed to be preempted by other memory phases. For a set of real-time tasks running under Memory-Centric Scheduling, we have proven a formal real-time schedulability bound [40].

An example scheduling trace using Memory-Centric Scheduling of one core of a system running three jobs, with a TDMA slot size of 2 time units, is depicted in Figure 3.13. In Figure 3.13(a), the three jobs are scheduled according to a fixed priority policy, with \( J_1 \) having the highest priority and \( J_3 \) the lowest. Note that the core stalls in intervals \([3, 4]\) and \([10, 12]\), since memory phases
can only run during the TDMA slot and there are no active execution phases. The trace in Figure 3.13(b) shows the same task set under Memory-Centric Scheduling. Memory promotion ensures that the memory phase of $J_3$ finishes at time 10, hence preventing stall in interval $[10, 12]$. Similarly, nonpreemptive execution of memory phases prevents stalling during $[3, 4]$ by allowing $J_2$ to complete first.

Memory-Centric Scheduling, however, is only one approach to scheduling PREM-compliant tasks. Is it the best approach? In the upcoming sections, we will use a simulation-based analysis to gauge the performance of Memory-Centric Scheduling compared with other reasonable candidates.

3.2.3.2 Abstract PREM Task Model

Before we perform our simulation-based evaluation of various scheduling algorithms for PREM-compliant tasks, we must first justify the task sets which are generated to perform the evaluation. In traditional real-time research, one would typically generate task sets by creating tasks with varying periods and execution times [57]. In our case, however, we are instead interested in evaluating schedulers for PREM-compliant tasks. For this reason, we must also come up with a reasonable way to generate PREM-compliant tasks, where each job’s execution is divided into various phases, some of which are memory phases and some of which are execution phases.

In order to make our simulation results meaningful, we want to perform this division in a responsible way, as opposed to haphazardly dividing each task’s execution time into various phases.
Therefore we need to first have an understanding of the way typical real-time applications behave when they are made PREM-compliant. For this purpose, we reexamine several benchmarks from the AutoBench group of the EEMBC (Embedded Microprocessor Benchmark Consortium) [58] benchmark suite which we had earlier made PREM-compliant [40]. Particularly, we are interested in extracting parameters from the benchmarks that we could use when generating task sets in our simulation-based multicore PREM evaluation in Section 3.2.3.4.

The benchmarks we examined were of four streaming applications which would process batches of input data and produce a stream of corresponding output. Specifically, the four benchmarks we used from the AutoBench group were a2time (angle to time conversion), canrdr (response to remote CAN request), tblook (table lookup) and rspeed (road speed calculation). Ideally more benchmarks would have been examined, however, making a benchmark PREM-compliant is currently a manual task which requires examining the behavior of the code in order to know where in the program cache misses may occur, and then inserting the appropriate prefetch macros in the memory phase which precedes the corresponding computation phase. We have already described a wider feasibility analysis of making tasks PREM compliant in Section 3.2.2.

Each benchmark in the EEMBC suite comes with a sample data file that represents typical input for the application. Since the original input data is of finite size, in order to evaluate the behavior of the benchmark under larger inputs, we replicate the original input data as often as is needed in order to have the desired input size. This approach, also used in other earlier work [46], is reasonable because the benchmark conceptually processes a stream of input data, instead of a fixed data set of a certain size.

Our measurement platform for this experiment was also an Intel Q6700 CPU platform, the same system used for our PREM implementation described earlier. This setup has a maximum theoretical memory bandwidth of 1.8 GB/s. However, due to the extra delay caused by cache write-back, the effective memory bandwidth during memory phases that we measured was roughly half of this, 819 MB/sec. The platform provides 1 MB of last-level cache per core.

For each benchmark, we measured the amount of execution time in CPU cycles in both the memory phases and execution phases. Since we have the flexibility of controlling the length of the predictable interval (by varying the input sizes), all the measured numbers are based on a single interval. As we increased the total input size, we observed that both the memory and execution phases grow linearly with the input data size. This is somewhat expected, since these benchmarks are of streaming tasks which process batches of small inputs.
Figure 3.14: As the size of the input is increased, the cache stall ratio quickly converges to around a different constant value for each benchmark.

From these measurements, we could also compute a critical parameter which describes each benchmark, the cache stall ratio. The cache stall ratio is defined as the ratio of time spent in the memory phase to the total time of the interval. Figure 3.14 depicts the cache stall ratio for all four benchmarks when the input data size is changed. When the total input is small, there is a fixed amount of both prefetch overhead and execution overhead. This is due to the prefetching of the code of the benchmark, as well as variables which do not scale in size as the size of the input changes, as well as a fixed amount of execution which is not dependent on the input’s size. When the input size increases, the phase times gradually stabilize to a linear trend with the input size. Different benchmarks show different cache stall ratios, in the range of 4% to 19%, as shown in the figure.

Our measurements on the EEMBC benchmark applications yielded two observations useful to help generate PREM-compliant task sets:

- For each streaming real-time task, both the memory phase time and the execution phase time are a function of the input size. These times can be predicted from the input size using a linear model with a fixed y-offset (which captures the input-independent prefetch and execution) and a slope (which captures the input-dependent prefetch and execution). For larger execution times the slope component dominates the execution time.

- The cache stall ratio, defined as the ratio of the memory phase to the total execution time, is constant for each benchmark when the input sizes are sufficiently large. Values for the cache stall ratio of typical applications can be between 4% to 19%.

The first observation is specific to streaming real-time tasks, since, of course, a general application can scale in drastically different ways as the input size is changed. However, since streaming real-
time tasks conceptually process lots of small independent inputs, modeling the phase times with a linear approximation is reasonable. The second observation is helpful in guiding the parameters we will use for simulation task set generation. Since the cache stall ratio dominates the execution time for anything but small input sizes, we can consider generating tasks by choosing a value of cache stall ratio in this range, and then generating appropriately-sized memory and execution phases.

3.2.3.3 PREM Simulator

A simulator for PREM-compliant tasks must be capable of simulating four key aspects of the PREM execution model. First, there are multiple cores which must be concurrently scheduled. Second, each periodic job consists of phases which prefetch data from main memory (memory phases) and phases which do CPU-only computation on cached memory (execution phases). Third, there is a single global shared resource, memory access, which all tasks compete for, and generally must only be accessible by one core at a time (otherwise, it should be possible to simulate the timing interference caused by concurrent access). Last, memory access is not a traditional nonpreemptable critical section, but can instead be preempted at any time. We would also like to generate task sets which match the information we derived from the EEMBC steaming real-time benchmarks in Section 3.2.3.2.

Due to these unique requirements for simulating PREM-compliant systems, we could not directly use an existing multicore real-time systems simulator, and instead created one which fulfilled the requirements. The two key components in our simulator are task set generators and CPU schedulers. In this section, we will describe the first of these. CPU schedulers are better described in the context of their corresponding simulation’s measurements in the next section. For this reason, we postpone their detailed explanation until then.

The simulator we created allows the generation and simulation of tasks which are PREM-compliant. The simulator base is written in Java and provides the framework to perform the simulation. The simulator uses a modular design where different algorithms can created without modifying the simulator base. The simulator base provides functionality to allow the user to select which task set generation algorithm and scheduling algorithm to use, and define the parameters for these algorithms, as shown in Figure 3.15. Additionally, simulation parameters including the number of cores in the system, the random seed to use, and the number of simulations or desired simulation time are provided by the user. When simulation begins, the simulator base spawns multiple threads based on the number of available processors on the machine it is run, and then, after the desired
number of simulations completes, reports schedulability results in a table. The user can then select an individual simulation from the report table, and examine a visual trace of the specific simulation run.

Individual simulations are run by considering all tasks to be released at time 0, and then running the simulation for one (system-wide) hyperperiod. If no tasks have missed their deadline at the end of the hyperperiod, the task set is reported as schedulable. Although this method captures neither the best-case nor the worst-case alignment of tasks in a multicore system, it can generally indicate which algorithms perform better than others, which is the main purpose of our evaluation.

A Task Partitioner in the simulator is responsible for generating task sets, and partitioning them onto processors. Each task partitioner can define a set of parameters and default values. The simulator base will then allow the user to modify these simulation parameters from the user interface. Additionally, each task partitioner must override a `createTasks` method which, given a list of cores, must create a task set according to the user-defined parameters and assign tasks to each core. For our evaluation, we created a parameterized task partitioner to create tasks with properties similar to the EEMBC benchmark tasks we examined previously in Section 3.2.3.2. The user provides several parameters, which can be defined as either specific values or ranges. The task period parameter, for example, is specified with a range and a quantum value. Only periods which are in the range and a multiple of the quantum are considered. This keeps the hyperperiod of the task set manageable, so that simulations can complete in a timely manner. The parameters generally varied between simulations, however, two were kept constant. For the cache size per core, we used our testbed (described earlier in Section 3.2.3.2) value of 1 MB. For the effective memory speed, we
used the measured value of 819 MB/s in all simulations.

Our EEMBC-based task partitioner works by first dividing the user-defined total utilization by the user-defined number of tasks to generate. This is done using UUniSort algorithm [57] with one slight modification. Since we are dealing with a multicore system, this approach can yield individual tasks with more than 100% utilization, which are never schedulable. In these cases, we run the UUniSort algorithm again to generate a new task set. Although this loop is not guaranteed to terminate, in our evaluation, task sets containing only tasks with less than 100% utilization were always found in a timely manner.

After the utilization was determined, each task is assigned a task period. This is done by uniformly picking a value from the task period range which is a multiple of the task period quantum. The execution time can be determined from the task’s utilization and the now-computed period. The total memory phase time is then determined from the cache stall ratio and the execution time. The remaining time (total execution time minus memory phase time), is the total execution phase time. One concern with this method is that applications would typically have varying phase execution times rather than a single fixed time. This can be alleviated in implementation by forcing each phase to always execute for its worst-case time [39], maintaining a predictable effect on the other tasks in the system.

Next, tasks are statically partitioned into cores. In our task partitioner, we use a worst-fit decreasing heuristic in order to do this assignment. This means that the task with the highest utilization is first assigned to the core with the lowest total utilization. Next, the task with the second highest utilization is put on the core with the new lowest total utilization. This repeated until no tasks are left. Other task partitioners could also have been considered, but we leave that for future research and instead focus more on the CPU schedulers.

In the last step of task generation, each task has the potential to be split into multiple pairs of memory/execution phases. This splitting is done by first determining the cache space available to each task. We evenly divide the core’s cache among tasks assigned to that core. The memory footprint of the task is determined using the total memory phase time and the memory access speed parameter. Then, we greedily create memory phases which fill the available cache space, followed by execution phases determined by the cache stall ratio. This is repeated until no execution time is remaining. Notice that this type of splitting is possible because we are dealing with streaming real-time tasks, where we have a choice of processing a large batch of data all at once, or several smaller batches in sequence.
In our simulator, a **CPU Scheduler** determines which tasks should execute at each point in time. Each CPU scheduler can define a set of parameters and associated default values. As with the task partitioner, the simulator base allows the user to modify these parameters from within the user interface. The simulator must also define two methods: `isSchedulable`, which returns a boolean if the particular set of tasks assigned to cores is schedulable, and `getSchedule`, which returns a complete trace of the system (which indicates which task executed at every point in time). The difference between these two methods is that the first can be optimized to simply check for schedulability without storing a trace (this is the one used when generating a report about a large number of simulations), whereas the second is used only when the user selects to view a trace of an individual simulation.

We implemented several classes of parametrizable CPU schedulers. These algorithms are better described, however, in the context of their simulations measurements in the next section. For this reason we postpone their detailed explanation until then.

### 3.2.3.4 Multicore PREM Evaluation

In order to evaluate various scheduling approaches in the PREM-compliant framework, we took a simulation-based approach. In this approach, we generated synthetic task sets using realistic parameters, and then used various scheduling algorithms to try to schedule the task sets. While this approach does not provide a worst-case scheduling bound, it does allow us to evaluate a large number of different algorithms. This is important since the space of reasonable algorithms is quite large, and a theoretical analysis of each one, along with a corresponding comparison with the others, would take colossal effort.

Using the simulator we had build, we could evaluate many scheduling strategies for PREM-compliant tasks. Even though simulations are quick to run, due to the sheer size of the search space, we still needed to limit the types of algorithms we considered.

We first will explain the details about the algorithms we considered and the measurements we performed. Our simulations were focused on two broad classes of schedulers, each with a wide selection of parameters. The first category are schedulers which are TDMA-based. These are a generalization of the previously-described Memory-Centric Scheduling approach from Section 3.2.3.1. We then focus on algorithms similar to the Simple Partitioned FIFO Locking Protocol (SPFP) algorithm [59]. In these algorithms, which we call prioritized memory schedulers, memory is treated as a global shared resource over which the tasks running on the cores compete. We then compare the
two classes of schedulers against each other and the case where no implicit scheduling is performed. Finally, we discuss the results and conclusions of our simulation study.

**Simulation Assumptions and Details.** Our evaluation will focus on schedulers which work on statically partitioned task sets. This means that algorithms like global RM or global EDF are not considered. We also assume that there is a static set of periodic tasks that we are trying to scheduling with task periods equal to deadlines. We also do not include shared resources among tasks, except for main memory, which all tasks will access.

We use the same task partitioner, the worst-fit decreasing one described in Section 3.2.3.3, for all of our experiments. This partitioner always evenly splits the cache among all tasks assigned to each core, and divides each task into as few memory phases as possible, in a greedy manner.

In the evaluation, we report either the fraction of tasks found to be schedulable at specific utilization levels, or something called weighted schedulability [60]. The weighted schedulability is defined as follows, given a set of evenly-spaced utilization levels $Q$, where $\text{fracScheduled}$ is a function which returns the fraction of schedulable tasks at a particular utilization level:

$$\frac{\sum_{U \in Q} U \cdot \text{fracScheduled}(u)}{\sum_{U \in Q} U}$$

The weighted schedulability is a number between 0 and 1 which indicates a particular scheduler’s ability to schedule tasks over a specific range of utilization levels. A higher weight is assigned to the value when the scheduler is capable of successfully scheduling task sets with a high utilization. It is important to be aware of the utilization range being considered when measuring weighted schedulability, since the final value is sensitive to the utilization range considered. A range which includes low utilizations with more schedulable task sets will inflate the final result, and a range which includes more high utilizations which are unschedulable will deflate it. For this reason, it is most useful when performing a comparison of multiple algorithms, rather than in absolute terms.

At each utilization level, we approximated $\text{fracScheduled}$ by performing 250 simulations on generated task sets, according to the parameters of the simulation. Thus, for example, if there are 100 samples in the utilization range, each value of weighted schedulability is the combination of 25,000 individual simulations.

In the simulations presented, unless otherwise stated, we will initially consider the following task set generation parameters. The system contains 4 cores. For each core in the system, 5 tasks are
created (20 tasks for 4 cores). Task periods chosen between 5ms and 8ms using a period quantum of 1ms. Tasks have a cache stall ratio between 15% and 20%. This is at the higher end of memory utilization that we expect, based on the measurements from the EEMBC benchmarks. Modifications to these parameters will be considered in the comparison section after our initial description of the algorithms we compare.

**TDMA-based Schedulers.** First, we describe experiments with various TDMA-based schedulers. In these schedulers, there exists a global TDMA memory schedule among cores, and each core is only allowed to access main memory when the core has its TDMA memory slot assigned. These schedulers are based on Memory-Centric Scheduling, as described in Section 3.2.3.1.

Memory-Centric Scheduling gives each core a fixed-size TDMA slot. When the core is granted memory access, all memory phases which are pending are promoted to a priority level above any execution phases (in order to make use of the memory slot). The intuition behind this effect, which we call *memory promotion*, is that memory is the bottleneck of the system, and so it should have priority over CPU-only computation. Additionally, once a memory phase begins executing, it will be run to completion before any other memory phases are serviced (if the TDMA slot ends, the memory phase will resume when the next TDMA slot arrives). The intuition behind this rule, called *nonpreemptive memory phases*, is that it allows more memory phases to run to completion, and therefore allows their corresponding execution phases to become schedulable. We will evaluate the two main rules of Memory-Centric Scheduling, memory promotion, and nonpreemptive memory phases, over various slot sizes.

For the comparison of TDMA-based algorithms, the weighted schedulability set \( Q \) contains utilizations in the range \([0, n]\), where \( n \) is the number of cores, with a utilization step size of \( n/100 \).

In the context of Memory-Centric Scheduling, we can consider which CPU scheduler to use. In the original work [40], Rate Monotonic was used as the CPU scheduler for computing the bound. However, we observed that using EDF to schedule tasks on the cores generally improved schedulability (which matches the theoretical bound [61] on individual cores). For this reason, all of our reported results are using EDF as the CPU scheduler.

In the context of TDMA-based scheduling, we evaluate the effect of each of the two rules of Memory-Centric Scheduling. We consider all four cases, with and without memory promotion, and with and without preemptive memory phases. The results are shown in Figure 3.16. Surprisingly, nonpreemptive memory phases have only a minor effect on schedulability with these task set gener-
Figure 3.16: Memory promotion has a much larger effect on schedulability than nonpreemptive memory phases.

ation parameters. The largest improvement to schedulability is provided by the memory promotion scheduling rule. The approach suggested by Memory-Centric Scheduling, using memory promotion with nonpreemptive memory phases, seems to be best even with the EDF CPU scheduler.

We now consider a modification to the TDMA-based scheduling where cores that do not have any pending memory phases can give up their TDMA slot early. In these systems, we expect a better utilization of memory bandwidth, since cores which have completed all their memory phases do not block memory if there are other cores which require memory access. Hence, the scheduling policy turns out to be round robin, with an enforced maximum time that an individual core can access memory continuously, if other cores need access. A comparison with and without the addition of the round robin rule is shown in Figure 3.17. As expected, the round robin rule increases schedulability.

One interesting observation about round robin scheduling is that, as the TDMA slot size increases, it does not degrade to the point where almost no task sets are schedulable (as happens when the round robin rule is not used). This is because at some point, the slot size becomes large enough that each core will execute until it has no more memory phases pending, and then the next core will be granted memory access. This makes the scheduler less sensitive to the choice of TDMA slot size.

For TDMA-based schedulers, the best approach we have found in terms of schedulability is to use Memory Centric Scheduling with EDF on the CPU, along with the round robin rule and a small TDMA slot size.
Figure 3.17: When using memory promotion with nonpreemptive memory phases, the round robin rule further improves schedulability.

Prioritized Memory Schedulers. Prioritized memory schedulers are the other class of PREM-based schedulers that we consider. These schedulers work fundamentally differently than TDMA-based schedulers. Here, each core executes according to a local scheduling policy (we use EDF). When a memory phase is encountered, tasks across all cores compete to get memory access. There is a global, high-level memory arbiter in place which determines which single task among all those with pending memory phases will be granted main memory access. If a task is granted memory access, its CPU scheduler always schedules that task immediately. If access is not granted, the memory-requesting task is blocked and another task, which may not need memory access, may execute on the same core. The global memory arbiter has the option to revoke memory access in the middle of a memory phase as well, and grant it to a different task.

Prioritized memory schedulers generalize the approach used by the Simple Partitioned FIFO Locking Protocol (SPFP) [59]. In SPFP, global resource contention is resolved by putting tasks which compete for global resources into a FIFO. Then, nonpreemptive access to the resource is granted in FIFO order. Unlike general global shared resources, however, memory access has the unique characteristic that it can be revoked at any time. This allows us to explore other types of strategies for resolving the case where multiple tasks compete for memory access. In addition to FIFO, we also considered other approaches for arbitrating memory access among tasks, including Rate Monotonic (RM), Earliest Deadline First (EDF), shortest-memory phase first, and least-laxity.
In the case of least-laxity-first, an ideal scheduler is not implementable. This is because job laxity changes as a task executes, which can cause an excessive number of preemptions. For this reason, we first consider an approximation of a least-laxity first scheduler which uses a scheduling quantum, \( q \). This scheduling algorithm will perform arbitration to decide which task should access memory at every scheduling event (task release, deadline, or completion of a scheduling interval), or after \( q \) time elapses, whichever comes first. In our first measurement, we consider a quantum-based least-laxity first scheduler where \( q \) is set to a small value, 25\( \mu s \). Although this is still likely unimplementable, it can still be used as a gauge to compare other algorithms against.

For these schedulers, we will also use the memory promotion rule, where memory phases are given higher priority on cores above CPU-only execution. Similar to the TDMA-based scheduling, this rule improves schedulability because CPU cores have more flexibility in terms of which tasks to schedule.

In partitioned multiprocessor systems, it is usually considered meaningless to compare the priority of two tasks on different processors, since one processor may be underutilized. However, since we are assuming a specific task partitioning algorithm, we did notice that comparing priorities across cores, for example as was done in the prioritized memory case for RM and EDF, did improve system schedulability.

We compared prioritized memory access schemes using a FIFO policy for memory access, as well as RM, EDF, shortest-memory phase first, and least-laxity first. The results, in terms of weighted schedulability ratio, are shown in Table 3.3. The utilization range used for the weighted schedulability computation was from the utilization value \( u_l \) at which one of the scheduling algorithms could no longer schedule 100% of the tasks, up to the utilization level \( u_h \) where none the schedulers could successfully schedule any of the simulated task sets. The utilization step size was kept constant at 0.025.

<table>
<thead>
<tr>
<th>Parameter Deviation</th>
<th>([u_l, u_h])</th>
<th>EDF</th>
<th>Laxity</th>
<th>RM</th>
<th>FIFO</th>
<th>Shortest</th>
<th>Figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>[2.05, 3.90]</td>
<td>0.60</td>
<td>0.77</td>
<td>0.50</td>
<td>0.50</td>
<td>0.54</td>
<td>-</td>
</tr>
<tr>
<td>8 Cores</td>
<td>[2.45, 5.70]</td>
<td>0.53</td>
<td>0.78</td>
<td>0.36</td>
<td>0.36</td>
<td>0.30</td>
<td>-</td>
</tr>
<tr>
<td>12 Cores</td>
<td>[2.85, 5.93]</td>
<td>0.64</td>
<td>0.88</td>
<td>0.35</td>
<td>0.34</td>
<td>0.32</td>
<td>-</td>
</tr>
<tr>
<td>16 Cores</td>
<td>[2.85, 5.93]</td>
<td>0.74</td>
<td>0.90</td>
<td>0.42</td>
<td>0.37</td>
<td>0.37</td>
<td>Figure 3.18</td>
</tr>
<tr>
<td>Nonpreempt Mem</td>
<td>[2.05, 3.90]</td>
<td>0.62</td>
<td>0.78</td>
<td>0.54</td>
<td>0.50</td>
<td>0.55</td>
<td>-</td>
</tr>
<tr>
<td>8 Cores, Nonpreempt Mem</td>
<td>[2.85, 5.93]</td>
<td>0.51</td>
<td>0.78</td>
<td>0.34</td>
<td>0.30</td>
<td>0.24</td>
<td>Figure 3.19</td>
</tr>
<tr>
<td>12 Cores, Nonpreempt Mem</td>
<td>[2.85, 5.93]</td>
<td>0.66</td>
<td>0.87</td>
<td>0.38</td>
<td>0.33</td>
<td>0.33</td>
<td>-</td>
</tr>
<tr>
<td>16 Cores, Nonpreempt Mem</td>
<td>[2.85, 5.93]</td>
<td>0.77</td>
<td>0.90</td>
<td>0.44</td>
<td>0.36</td>
<td>0.39</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.3: Weighted Schedulability of Prioritized-Memory Algorithms
Figure 3.18: In high-contention situations, EDF and least-laxity first prioritized memory scheduling perform best.

Figure 3.19: Having nonpreemptive memory phases improves the schedulability for both EDF and least-laxity first prioritized memory scheduling.
As the number of cores increases, so does the contention for memory access (and the number of tasks competing for access, since we keep the number of tasks per core constant). For this reason, the difference in schedulability is even more visible when the number of cores is larger. In order to gauge the absolute difference between the algorithms, a graph of the fraction of schedulable task sets for each utilization level in a 16 core system is shown in Figure 3.18. Here, the best algorithms for prioritized memory access are EDF and least-laxity first.

As mentioned before, a least-laxity scheduler, even the quantum-based version we considered, can cause a huge number of preemptions during a memory phase which makes it not implementable. We now consider the effect of making the memory phases nonpreemptive.

Table 3.3 contains the weighted schedulability for the version of the schedulers where memory phases are made nonpreemptive. For memory access which is prioritized by EDF and least-laxity first, we also provide a comparison with the preemptive versions in Figure 3.19. Here, both schedulers benefit from nonpreemptive memory phases. However, this modification allows for the better of the two algorithms, least-laxity first, to be realizable in an implementation.

Comparison. We now compare the best schedulers we have found so far from both categories. Additionally, we considered the unscheduled case were concurrent memory access causes timing interference among cores. We also consider a wider range of task generation parameters to make sure our analysis is not subject to the specific parameters we have been using in the simulations so far.
Figure 3.20: A comparison with the regular task generation parameters shows M-LAX performing best.

From TDMA-based schedulers we consider the TDMA-Round Robin (TDMA-RR) scheduler with a small slot size (25\(\mu s\)). Although a 25\(\mu s\) slot size is likely unimplementable due to the context-switch overhead, it serves as an upper bound on the performance that can be expected from a TDMA-based approach. From prioritized memory schedulers, we will evaluate least-laxity first with nonpreemptive memory phases (labeled as M-LAX) as well as an EDF-based scheduler with nonpreemptive memory phases (labeled as M-EDF). We also include a contention-based scheduler, which represents the unscheduled case where memory phases among different cores can overlap. In contention-based scheduling, task execution is slowed down whenever there are concurrent memory phases on different cores. When \(n\) cores have overlapping memory phases, contention-based scheduling fairly divides the memory bandwidth among tasks so each core gets \(\frac{1}{n}\) of the original memory bandwidth (resulting in a longer memory phase). This would happen if the low-level hardware arbiter used some form of round-robin and the memory requests were of equal size. This is, however, slightly pessimistic in that it assumes memory phases are always constantly accessing memory.

A comparison with the original scheduling parameters is given in Table 3.4. Among the scheduling algorithms we have investigated, M-LAX outperform the others, including the contention-based case. A view of the absolute difference with 12 cores is shown in Figure 3.20.

We modified the cache stall ratio to show the effect of the amount of memory contention on the different scheduling approaches. As seen in Table 3.4, when contention is low, all schedulers do comparatively better (the utilization ranges used for weighted schedulability, and the value of
Figure 3.21: As we increase the number of tasks in the system, contention-based scheduling lags behind the other approaches.

weighted schedulability are both higher). As the cache stall ratio increases, as the tasks demand more memory access, M-LAX begins to outperform the other schedulers by an increasing margin.

Next, we increase the number of tasks in the system. A representative plot is shown in Figure 3.21. The number of tasks per core is used to determine the number of tasks in the entire system. However, due to the worst-fit-first decreasing task partitioner, this does not mean that each core has the same number of tasks assigned to it. With more tasks in the system, at the same utilization level, each task generally will have a smaller utilization. By increasing the number of tasks at a fixed utilization level, all schedulers experience improvement in schedulability. Contention-based scheduling lags behind the others in the amount of improvement, however. This matches the result from the Memory-Centric Scheduling bound [40]. As we add more tasks to the system at a fixed utilization level, the length of the memory phases becomes shorter. The TDMA-RR approach does well in this situation, outperforming even M-LAX. Although M-LAX still does well in this case, TDMA-RR more evenly divides bandwidth among the cores, which is important in this situation.

When we modified the task periods to be between 1ms and 10ms with a period quantum of 1ms, as shown in Figure 3.22, M-LAX performs best. This shows it is a good choice for other task periods as well. TDMA-RR performs poorly here because a fair bandwidth allocation does not take into account the larger difference in task criticality.

**Discussion.** Among the schedulers we have investigated, M-LAX scheduling, which is scheduling
where memory phases are arbitrated globally using a nonpreemptive least-laxity first approach, generally performs best. Even in the case where there are more tasks per core, and TDMA-RR does best, M-LAX is not far behind. In these cases, the TDMA-RR we simulated, with a $25\mu s$ TDMA slot size, is an upper bound on the expected performance of an implementable TDMA-RR with a larger, more realistic slot size. Having a larger slot would likely degrade performance, as shown in Figure 3.17.

We believe a laxity-based approach works better than EDF when choosing which tasks to run in a multicore system. This is because individual tasks can vary greatly in their utilization, so a task with a long deadline on one core might actually have a long execution time as well, so it cannot be excessively delayed. If another core has a task with a shorter deadline, however, it would get priority in an EDF-based approach. Using a least-laxity first scheduling rule is better because it takes into account not just the deadlines, but also the remaining execution time.

The nonpreemptable aspect of memory phases in M-LAX scheduling is not only necessary because a fully preemptable version is unimplementable, but this rule actually improves schedulability (Figure 3.19). This is due to the case when tasks on different cores running memory phases have identical laxity. A preemptable least-laxity scheduler would switch back and forth between them, so only one is executing at a time. The nonpreemptive version will execute an entire memory phase, and then, when the second task's memory phase is running, the first task can run its execution phase. This improves parallelism, which generally benefits schedulability.
One potential downside of nonpreemptive memory phases is that they can cause priority inversion. This can especially cause problems when the memory phase times approach the lengths of the periods of tasks, since the maximum memory phase time represents the maximum blocking time from this priority inversion. For this reason, we do not expect M-LAX to do well when the minimum period in the system is small, where small means comparable to the size of the memory phases in the task set. This can be somewhat resolved by, if the task permits it, making memory phases even smaller, or splitting memory phases into multiple nonpreemptive chunks. This splitting, however, will likely increase task preemptions.

In conclusion, our evaluation indicates that partitioned PREM-compliant task sets are generally best scheduled by an M-LAX scheduler. This scheduler uses EDF to schedule computation on each CPU, but promotes memory phases above CPU-only computation. Memory arbitration among tasks is done through a global high-level arbiter which chooses the memory phase corresponding to task with the least laxity. Surprisingly, the nonpreemptive memory phases in M-LAX serve not only to make the scheduler implementable, but also were shown to generally outperform a quantum-based strategy for least-laxity approach that approximates the fluid model.

3.3 Related Work

Real-Time I/O Management System. Several related works address the problem of interference while accessing main memory. Some approaches look at analysis of interference among mutually-transmitting peripherals [62, 63]. Empirical approaches can estimate the impact of PCI-bus load on task computation time based on experimental measurements of reference tasks [64]. Alternately, analytical approaches exist to bound I/O interference [65]. However, the analysis is restricted to a single DMA controller using predictable cycle-stealing arbitration, and can not be applied to a COTS system. There is also analysis to estimate the impact of mutual interference among processing cores. For example, static analysis can compute cache access delay in a multiprocessor system [66]. However, these results focus on deriving the increase in task execution time while neglecting the effect of delay on communication flows.

Modeling complex COTS interconnections and estimating delay and buffer requirements for peripheral flows can be done in an AADL-based environment [67]. An event-based model may be used to estimate delay for both computation and communication activities in a multicore system-on-chip [28]. However, lack of precise knowledge of COTS behavior implies that this analysis must
make pessimistic assumptions, which can lead to high delay and buffer sizes. The Real-Time I/O Management System removes such unpredictability by forcing an implicit bus schedule.

Several other analysis frameworks exist to estimate delay characteristics for communication flows in embedded systems. For example, Real-Time Calculus can compute end-to-end delay for various real-time systems [68, 69]. Analysis methodologies are available for existing real-time interconnections such as CAN [70]. However, these methodologies typically assume a detailed knowledge of each component’s behavior, which is often unavailable in a COTS-based system.

**Predictable Execution Model.** Prior real-time research has proposed several solutions to address different sources of unpredictability in COTS components, including real-time handling of peripheral drivers, real-time compilation, and analysis of contention for memory and buses. For peripheral drivers, Facchinetti et al. [71] proposed using a non-preemptive interrupt server to better support the reusing of legacy drivers. Additionally, analysis can be done to model worst-case temporal interference caused by device drivers [72]. For real-time compilation, a tight coupling between compiler and worst-case execution time (WCET) analyzer can optimize a program’s WCET [73]. Alternatively, a compiler-based approach can provide predictable paging [74]. For analysis of contention for memory and buses, existing timing analysis techniques can analyze the maximum delay caused by contention for a shared memory or bus under various access models [28, 29]. All these works attempt to analyze or control a single resource and obtain safe bounds that are often highly pessimistic. Instead, PREM is based on a global coschedule of all relevant system resources.

Instead of using COTS components, other researchers have discussed new architectural solutions that can greatly increase system predictability by removing significant sources of interference. Instead of a standard cache-based architecture, a real-time scratchpad architecture can be used to provide predictable access time to main memory [75]. The Precision Time (PRET) machine [76, 77] promises to simultaneously deliver high computational performance together with cycle-accurate estimation of program execution time. Unfortunately, these solutions require extensive redesign of existing components (in particular, the CPU). While our PREM execution model borrows some ideas from this work, it is compatible with available COTS platforms: all existing components can be reused, albeit some new devices must be connected to the motherboard and the COTS peripherals. This approach allows PREM to leverage the advantage of the economy of scale of COTS systems, and support the progressive migration of legacy systems.

**Multicore PREM Scheduling.** In terms of real-time multiprocessor scheduling, many solutions have been proposed [78], but most results do not consider the interference caused by shared
physical resources like caches, buses, memory. Many timing analysis frameworks have been proposed to capture such interference, but unfortunately, WCET estimations tend to be quite pessimistic. Hence, instead of performing timing analysis, we focus on memory and cache-aware scheduling techniques. Holman et al. [79] reduced bus contention at the start of the scheduling quantum caused by simultaneous data reloading into the cache. Anderson et al. [80] provided methods to reduce cache contention in a multicore system with a shared cache. However, these approaches are not optimized for the worst case, even though they perform well in the average case. Other solutions have been developed for soft real-time systems [81, 82, 83, 84], but they cannot be directly used to provide guarantees necessary for hard real-time. Multicore systems with a shared L2 cache were also studied by Guan et al. [85], who proposed a scheduling algorithm and a schedulability test based on linear programming. Rosen et al. [66] proposed a method to obtain efficient TDMA arbitration policies by using static analysis.

Other research has investigated hardware-based architectural modifications to increase system predictability by removing the sources of inter-core interference. Such efforts include predictable DRAM controllers [86, 87, 88], real-time scratchpad architectures [75], and predictable pipelines [76]. Our work, in contrast, considers using COTS hardware, but with modified software which enables high-level scheduling of memory access to prevent contention at the low-level, non-real-time, hardware arbiters. In the PREM scheduling problem, main memory is essentially treated as a resource shared among all cores. In other words, to execute a memory-intensive portion of the code (memory phase), the task must acquire both its local processor as well as the shared memory resource. In this sense, the PREM scheduling problem is a special instance of the more general problem of real-time resource sharing in a multiprocessor. Several solutions have been proposed for this problem [89, 59], based on locking protocols for real-time multiprocessor systems [90, 91].

Brandenburg et al. [59] precisely defined the concept of blocking time in multiprocessor scheduling and presented optimality results for resource sharing protocols. In particular, a distinction is made between the case where a high-priority task waiting for resource access still allows lower-priority tasks to execute in its place (s-aware blocking), and the case where this is not possible (s-oblivious blocking). Since, in the PREM scheduling problem, tasks which are waiting for main memory access are suspended, we naturally consider s-aware blocking. The Simple Partitioned FIFO Locking Protocol (SPFP), which uses a single global FIFO for all shared resources, is asymptotically optimal in the s-aware case for partitioned scheduling [59]. Therefore, when evaluating multiprocessor PREM schedulers, we compared our approach with SPFP in Section 3.2.3.4.
3.4 Conclusions

In this chapter, we addressed the challenge of affordability in real-time systems. This was done by leveraging on the low cost and high performance nature of commercial-off-the-shelf (COTS) components. COTS components, however, often can not guarantee real-time requirements. We therefore proposed minimal additions to COTS systems and modifications to current practice, in order to make systems capable of satisfying real-time computation requirements.

We described two specific techniques to do this. One was a Real-Time I/O Management system, which could buffer incoming I/O traffic and deliver it predictably to main memory. The second was the PRedictable Execution Model (PREM), which modified software tasks running on the CPU to only access main memory at known intervals of time.

The high-level link between these approaches is the idea of implicit scheduling. Since the low-level bus and memory arbiters are not real time and not changeable, we instead control when the requests to shared resources occur. If only a single request to shared resource occurs at a time, it will be immediately granted, contention-free, by the non-real-time arbiter. We have demonstrated that implicit scheduling is effective for both I/O peripherals and for real-time software tasks.
Chapter 4

Node Safety

In cyber-physical systems, it is essential that the low-level controllers on the individual nodes, which actuate objects in the physical world, meet formal safety requirements regarding this actuation. Furthermore, the high-level safety approach outlined earlier in Chapter 2 relies upon a behavior model of the individual nodes of the distributed control system. If this model does not reflect the behavior of the individual nodes, errors in the low-level controllers may compromise not only local safety, but can also lead to violations of global safety properties of the CPS. Directly guaranteeing the node’s behavior through formal verification, however, is often impossible due to a lack of accurate models, or, even if such models are available, leads to an intractable verification problem. In this chapter, we will fill in this remaining gap of guaranteeing the behavior of the individual nodes, without mandating direct verification or exhaustive testing.

The approach we advocate is based upon the Simplex Architecture [18]. Simplex incorporates supervisory control and switching logic on top of an unverified controller, for the purpose of improving system robustness. A Simplex system consists of three main components, shown in Figure 4.1. Under normal operating conditions, the unverified complex controller actuates the system. If the system state becomes in danger of property violation, the safety controller takes over. After some time, the safety controller should drive the system to a state that can tolerate aggressive action without danger of property violation, and the complex controller is allowed to resume control. The switching between the controllers is performed by a decision module.

The advantage of the Simplex design is in the potential for simpler verification. By verifying the safety controller and decision module, properties about the composite system can be proven. Thus, we avoid having to perform verification on the complex controller directly, which, in some instances, can significantly reduce verification effort. For example, if the complex controller is hand-coded, models developed independently will not capture its true behavior (hand written code almost certainly will contain bugs), and accurate models generated from the code are often intractably large. Creating the safety controller in a verified manner can often be less intensive than creating
Figure 4.1: The Simplex Architecture provides system verification by using an unverified complex controller combined with a verified backup safety controller and verified decision module which chooses which controller to run at each point in time.

the complex controller, as only safety-critical considerations need to be taken into account, whereas the complex controller must also meet mission-critical requirements.

From the design of Simplex, it is clear that a correct decision module is essential. Some systems can not modeled easily with differential equations or have safety controllers that are too complex to permit proofs for useful properties [92]. In these instances, best practices from software engineering may be used to develop the decision module.

In other systems, a formal approach is possible. For verifying a decision module in a system with purely continuous dynamics, a Lyapunov function’s stability region can be used drive the decision module [93, 94]. A Lyapunov function for a controller for a particular plant defines a region of space where, if the current system state is within the region and the controller is used, the system will remain within the region indefinitely. If a Lyapunov function exists for the safety controller where the stability region is defined within the operational constraints without containing property violation states, and we know the maximum gradient over time for any controller within the Lyapunov stability region, we can formally derive the decision module switching rule. If our current system state is at least the maximum gradient times the control iteration time away from the Lyapunov stability region boundary, the complex controller can be used. In the worst case, the state space will proceed at the maximum gradient towards the Lyapunov stability region boundary, but will not cross it before the next control iteration. We can then switch to the safety controller which is guaranteed to stay within the Lyapunov stability region. In this way, the property violation region is never entered. The situation is shown in a two-dimensional state space in Figure 4.2. In practice, a margin larger than the maximum gradient times the control iteration time is often used to account for modeling errors and provide additional safety. After the safety controller returns
Figure 4.2: In a continuous two-dimensional state space, a Lyapunov function of the safety controller can be used to guarantee safety. The distance $d$ is at least the maximum gradient of the state space within the stability region times the control iteration time. If the state is within the grey region, the complex controller can be used. Otherwise, the safety controller is used.

the state to within the complex controller region, the complex controller can resume driving the plant. Typically, however, a smaller region is used when switching back to the complex controller, creating a hysteresis which prevents frequent controller switching. The drawback of this approach is that, even within strictly continuous systems, there is no general technique to determine a Lyapunov function for a particular controller and plant.

This chapter advances research in the Simplex Architecture in two key areas:

- Existing application-level fault-tolerance methods, even if formally verified, leave the system vulnerable to errors in the real-time operating system (RTOS), middleware, and microprocessor. We first introduce the System-Level Simplex Architecture in Section 4.1, which uses hardware/software co-design to provide a fail-operational guarantee despite such errors.

- Formal design of Simplex has previously mainly been concerned with purely continuous systems. Chapter 4.2 develops formal approaches to apply Simplex to hybrid system models, which consist of continuous dynamics defined by differential equations, and discrete logic represented by a finite state machine.

4.1 The System-Level Simplex Architecture

The original Simplex Architecture (which we refer to in this section as the Application-Level Simplex Architecture), is an approach to protect systems from faults in an unverified complex controller’s logic by providing a verified safety controller and verified decision module. However, most systems
in use will contain a set of dependent layers on which all our software relies for correctness. These include things like the microprocessor, the Real-time Operating System (RTOS) or the middleware. If errors are present in these components, or errors arise from their upgrades, the Application-Level Simplex approach can not guarantee safety.

For this reason we propose the System-Level Simplex Architecture, which provides robustness in the presence of both bugs in the application and bugs in previously dependent layers such as the RTOS. In this new architecture, we perform hardware/software partitioning on the Simplex framework. The two Simplex safety-critical components, the safety controller and decision module, are moved to a dedicated processing unit, not for the typical HW/SW co-design reasons of power and performance, but instead to provide isolation from software-related complexity. Additionally, this architecture meets the temporal constraints of the monitored safety properties by design.

In this section, we will first explain the key design elements of System-Level Simplex. We will then elaborate on the fault model under consideration. Next, we apply System-Level Simplex to two case studies. In the first, we examine pacemaker design and contrast the failures handled by the System-Level Simplex Architecture pacemaker with a previous-generation pacemaker. Then we apply the System-Level Simplex Architecture to an inverted pendulum system and verify its robustness with fault-injection testing. In our earlier research on System-Level Simplex [2], we had also created a design process to help produce both a valid architecture design and a valid implementation of a system which uses System-Level Simplex.

4.1.1 System-Level Simplex Design

The System-Level Simplex Architecture is based on the original Simplex concept[18]. As described in the chapter's introduction, Simplex is logically divided into three subsystems: safety, complex, and decision. The safety subsystem has a simple, reliable controller which provides verifiably safe performance. This is used in case the complex controller malfunctions. The complex controller drives the system as long as it does not jeopardize system liveness. This controller can be changed and upgraded while the system is running and may even contain bugs. The decision subsystem chooses which of the two previously-mentioned controllers to use. The decision module and safety controller make up the trusted computing base, and must function correctly for the system to remain safe, while most of the system's complexity is contained within the complex controller.

The Application-Level Simplex design had all three subsystems located in software. This worked well for protecting the system from value faults from the complex controller, however it required
that, to guarantee system safety, the middleware, the operating system, and the microprocessor were all fully reliable. We relax this requirement in the System-Level Simplex architecture by performing hardware/software partitioning on the system. The two Simplex safety-critical components, the safety controller and the decision module, are moved into a dedicated processing unit outside the microprocessor.

This is akin to hardware/software co-design, except that we perform this move not primarily for reasons of performance and power consumption, but instead to protect from software-related faults. The designer therefore has a choice of what the dedicated processing unit should be. One option is to use a microcontroller to run the two safety core subsystems. However, in safety-critical systems, even processors are not completely trusted [95], and we can choose to eliminate this underlying complexity. For example, we can run the Simplex safety core on dedicated hardware. Ideally we would produce an Application-Specific Integrated Circuit (ASIC), but instead, for cost and reprogrammability reasons, we opted to perform our evaluation using a Field Programmable Gate Array (FPGA). The same VHDL code used to program an FPGA can be used to produce an ASIC.

By moving the Simplex safety core to isolated hardware, we can also provide temporal correctness for the monitored safety properties. If the high-performance complex subsystem does not produce a control command in the appropriate time, whether caused by an RTOS bug, poor cache performance or excessive bus contention, the conservative safety controller’s output is used. Since the safety controller runs in parallel on isolated hardware (which prevents run-time variations caused by resource sharing), the temporal constraints are met by design.

4.1.2 Fault Model

The System-Level Simplex Architecture tolerates two broad categories of faults: logical faults and resource sharing faults.

Logical faults occur when the complex controller passes an unsafe value to the decision module, or a value of an incorrect type. One cause for this sort of fault is a malfunctioning complex controller. An incorrectly-typed value, on the other hand, may cause logical operations that use it to fail. For example, if the control commands are IEEE floating-point values that correspond to voltages, the values NAN or infinity are incorrectly-typed. Another logical fault occurs when a non-functional complex controller does not output any value.

Resource sharing faults are caused by failures in common resources among components. The
The original Application-Level Simplex Architecture shares several resources, each of which can cause the system to fail. These include all the physical and logical resources managed by the OS like memory, CPU, and shared libraries. These faults can manifest directly (a misimplemented library causing the decision module process to crash), or indirectly (a mismanaged processor causing timing faults). Additional shared resources may include the communication bus and the power source.

There also exist out-of-scope faults which System-Level Simplex does not address. Specifically, the sensors and actuators used by our system must be reliable and accurate. The FPGA hardware, which runs our Simplex safety core, is assumed to be correctly manufactured. Additionally, the synthesis process, which takes our VHDL code and generates FPGA bitstreams, is unaddressed. However, these faults are rare since companies strive to provide reliable hardware and synthesis tools, and some of them may be even further reduced by techniques like triple modular redundancy (TMR) [96]. We also do not handle environmental modeling faults which can be present in any system that uses formal methods. Since model checking is performed on the models and not the physical environment, a significant mismatch between the two results in an unsafe system, even if it is fully model checked. To account for these errors, the formal model should be reviewed, and fault-injection testing should still be performed on the final system.

### 4.1.3 System-Level Simplex Design for a Cardiac Pacemaker

A cardiac pacemaker is a piece of hardware inserted into a patient’s body in order to regulate his or her heart rate. Detailed designs of cardiac pacemakers have been disclosed [97, 98]. In this safety-critical application, we examine the practicality and usefulness of the System-Level Simplex Architecture, as well as the end-to-end design process.

We investigate two considerations for using the System-Level Simplex Architecture:

- Can the system be divided up into a safe controller and a complex controller, such that the most likely causes of failure are contained in the complex controller?

- How do the resultant safety guarantees compare to those of existing pacemakers?

#### 4.1.3.1 Dividing the Cardiac Pacemaker System

The first concern, the division of the system into complex and simple controllers, asks if the logical Simplex framework can be applied to a cardiac pacemaker. Since this is domain-specific, we examine some properties of artificial pacemakers.
The first generation of artificial pacemakers actuated the heart at a set interval. This functionality was sufficient to keep the patient alive, however, problems did arise. For example, when a healthy person walks up stairs or performs strenuous action, his heart rate increases. The first generation of pacemakers did not take this into account and patients would become dizzy and uncomfortable. Additional functionality was added to pacemakers to detect if the heart rate should be increased by monitoring the temperature of the blood, or the acceleration on the patient’s body [97]. Requirements were then added on top of this to preserve smooth heart-rate transitions, rather than suddenly jumping from 65 to 120 beats per minute because of a sudden large acceleration. Additionally, modern pacemakers attempt to detect and log anomalous events related to the heart to aid a doctor’s diagnosis. The logged data must be retrieved, and this is done through wireless communication with an external device.

Modern pacemakers have many other requirements, however we already covered enough to apply the System-Level Simplex Architecture. The rate-adaptive pacing modes, where the heart rate changes over time, require complex functionality. The pacing rate to which we should change is a function of the current rate, as well as the past and present accelerometer readings. The safety properties we want to enforce are that the heart rate should be between a lower rate limit and an upper rate limit, and should not change by more than a doctor-specified rate-smoothing parameter. These are the properties monitored by the decision module. The safe controller is a finite-state machine that meets the safety requirements. We choose a safe controller that slows down the heart to the resting rate (lower rate limit) in a way that satisfies the rate smoothing requirement. This safety controller does not have the complex rate-adaptive functionality, but instead provides a fail-operational mode that will maintain safety for the patient.

4.1.3.2 Comparing against Existing Pacemaker Reliability Mechanisms

The second consideration compares the existing reliability mechanisms found in one previous-generation pacemaker description [97] to the design created using the System-Level Simplex end-to-end design process. We focus on two mechanisms for enhanced reliability which were present in the pacemaker description we examined.

The first is a watchdog timer which is periodically reset during normal system execution. If the execution hangs at some point, the timer will not be reset and will timeout. The timeout triggers a high-priority interrupt which signals that an anomalous event has occurred and the system is reinitialized. Alternately, the system can be shut down as a fail-safe mechanism.
The watchdog timer mechanism is compatible with the System-Level Simplex Architecture. It provides a means to restart the system when it enters a rare error state. However, the watchdog timer does not protect the system from unsafe pacing, only system hangs. Additionally, deterministic bugs in the program will continue to restart the system, whereas a System-Level Simplex system is able to function safely in spite of deterministic bugs in the complex controller.

The other safety mechanism we examine is a redundant pacemaker system which, at first, appears to be similar to the Simplex approach. This system provides a simpler pacing mode without rate-adaption. This system takes control from the microprocessor when “a fault is detected in the operation of the microprocessor circuit.” This component, like the System-Level Simplex Architecture, provides protection from microprocessor errors. This is a real cause of concern with this specific pacemaker design because it uses a custom pacemaker-specific microprocessor. However, it does not provide protection from logical faults in the software. Additionally, control is switched to this system when any fault in the microprocessor is detected. In the System-Level Simplex Architecture, a hardware fault that only affects the logging mechanism (perhaps because of a rarely used instruction), one that does not compromise safety, would not trigger a change in control.

4.1.4 System-Level Simplex Design for Inverted Pendulum

An inverted pendulum is a classical control testbed where a rod must be maintained upright by moving a cart along a track. An inverted pendulum presents an obvious failure state when the rod falls over. We applied the System-Level Simplex Architecture to an inverted pendulum and evaluated its robustness by inserting faults and observing system robustness.

We guaranteed safety in this system through the same technique as previous Simplex applications [18]. By measuring the inverted pendulum system, we could define a Lyapunov stability function [99]. From this, we can generate the safety controller C code using Matlab Simulink [100] and determine when the decision module should switch controllers (before the state leaves the Lyapunov stability neighborhood). The C code was then manually translated to VHDL for hardware synthesis.

Our hardware safety core resides on an externally-powered Xilinx ML505 FPGA. This FPGA contains a PCIe port which is used to communicate to a PC which runs the software portion of the architecture. The software portion uses a custom driver written for Linux. We run Linux/RK [101] as the operating system for the complex controller. Since the System-Level Simplex Architecture handles timing faults, we purposefully do not use the provided real-time scheduler. Through memory-mapped I/O, the complex controller reads the most recent angle and track position and
<table>
<thead>
<tr>
<th>Failure Type</th>
<th>Safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Output</td>
<td>✓</td>
</tr>
<tr>
<td>Maximum Voltage</td>
<td>✓</td>
</tr>
<tr>
<td>Wrong Way — Maximum Voltage</td>
<td>✓</td>
</tr>
<tr>
<td>Time Degraded Control</td>
<td>✓</td>
</tr>
<tr>
<td>OS Crash</td>
<td>✓</td>
</tr>
<tr>
<td>Timing Faults</td>
<td>✓</td>
</tr>
<tr>
<td>Computer Reboot</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.1: The System-Level Simplex inverted pendulum implementation tolerates a variety of faults.

suggests a motor voltage to the hardware-based decision module.

After constructing the system, we verified that the software-based complex controller was able to actuate the inverted pendulum as long as it did not jeopardize safety. When the pendulum’s state approached the edge of the Lyapunov stability neighborhood, the safety controller took over and prevented the pendulum from collapsing. In this way, the system was able to tolerate a multitude of faults as outlined in Table 4.1. We outline two of these faults in detail.

4.1.4.1 Wrong Way — Maximum Voltage

The pendulum should remain balanced even if the complex controller outputs a motor voltage that would normally destabilize the system. This test took an extreme case of this where we used a working inverted pendulum controller for a few seconds, and then output the maximum voltage in the direction opposite of that needed to stabilize the pendulum. The decision module detected this and switched control to the safety subsystem. The safety controller returned the pendulum to a stable state and control was again given to the complex subsystem after a few seconds. Measurements from one iteration of this process are given in Figure 4.3.

4.1.4.2 Computer Reboot

The System-Level Simplex Architecture provides protection from arbitrary operating system behavior, including rebooting the system. From the decision module’s perspective, the computer rebooting is equivalent to a complex controller that sends no output. We ran this test on our inverted pendulum setup, and the pendulum remained stable throughout the reboot process. Additionally, after the computer restarted, the software-based complex controller was able to regain control of the inverted pendulum using memory-mapped I/O with the FPGA. This is significant because a common remedy for software problems is rebooting the computer. A malfunctioning complex controller can be repaired in this fashion while the physical system remains safe and stable.
4.2 Simplex Decision Module Generation

A system which uses the Simplex approach, whether it is the original Application-Level Simplex design or the System-Level Simplex architecture described in the previous section, needs a correct decision module to provide system safety. Existing formal approaches for this crucial task, as discussed in this chapter’s introduction, focus on purely continuous systems. We will now describe an approach for formal design of the Simplex decision module which is generalized for hybrid systems, which contain both continuous and discrete dynamics. This approach is based on computing reachability and time-bounded reachability, which will describe next. Finally, we will describe two tools we have created which use our research results in order to support Simplex-based design.

In the following sections we will consider hybrid systems, which are systems that contain both discrete and continuous dynamics. The basic approach we will describe uses reachability and time-bounded reachability in order to generate the decision module logic. This is described in Section 4.2.1. In the context of computing reachability, we consider general nonlinear hybrid systems, and develop an initial approach based on quantization in Section 4.2.2. Afterward, we show how we can increase accuracy of the approach arbitrarily in exchange for computation time, and apply it to a more general class of systems in Section 4.2.3.

4.2.1 Decision Module Generation Approach

In this section, we give briefly outline hybrid systems (a more thorough treatment will be provided later) and model Simplex as a hybrid automaton. With this modeling in place, the decision module logic can be expressed using the reachability and time-bound reachability of hybrid automata. In
the next sections, we will expand upon algorithms to compute this reachability and time-bounded reachability.

The key components for Simplex (shown in Figure 4.1) are (a) the complex controller (cc), (b) a safety controller (sc) and (c) a decision module (dm). Once every Δ time, the dm observes the plant and makes a decision about which controller (sc or cc) to activate. Roughly, if there is a possibility of entering an unrecoverable state within the next Δ interval, then dm activates sc. Once sc restores the plant to a state from which there is no possibility of violating safety in the next Δ interval, it reactivates cc. We model each controller/plant combination in the Simplex system as a hybrid automaton [102, 8, 9].

Informally, a hybrid automaton is a combination of differential equations with a finite state machine, where the state of the system can evolve both continuously and discretely. The continuous evolution of a hybrid system typically models the evolution of the physical variables in the plant, while the discrete transitions typically model software behavior. A state of a hybrid automaton consists of an assignment to each continuous variable, as well as the discrete mode of the automaton. The reachable set of states from an initial state are the states which the hybrid automaton may enter as time elapses, according to the switching rules and invariants of the automaton. A more formal presentation of hybrid automata and their semantics will be provided in later, in Section 4.2.2.1.

Let \( \mathcal{U} \) be the set of unsafe states of the plant model. \( \text{BackReach}(\mathcal{U}, \text{sc}) \) is defined as the set of states from which \( \mathcal{U} \) can be reached within the sc hybrid automaton. In order to drive the Simplex switching logic, we will use the set of states from which \( \text{BackReach}(\mathcal{U}, \text{sc}) \) can be reached in up to Δ time within the cc hybrid automaton. In the course of our research [6], we have shown that if sc is activated when the plant state is first detected to be in this set of states, then the overall Simplex system remains safe. Intuitively, this is for the following reason. The set of states \( \text{BackReach}(\mathcal{U}, \text{sc}) \) represent the unrecoverable states that may not be entered, since even the safety controller may reach an unsafe state from these. At each decision point, which happens every Δ time, we then will check if the complex controller can be used for the next Δ time. This amounts to checking if the Δ-reachable states (the states reachable in up to Δ time) from the current state contain any of the unrecoverable states. To compute these states directly from the unsafe states \( \mathcal{U} \), we use the expression:

\[
\mathcal{G} = \text{BackReach}_{\leq \Delta}(\text{BackReach}(\mathcal{U}, \text{sc}), \text{cc})
\]

Given this expression we can, in principle, generate a dm that is correct by construction. Un-
fortunately, the problem of computing backwards reachable sets for hybrid systems has been well studied and is, for general hybrid systems, undecidable [103]. Furthermore, an accurate model for cc may not be available for the reasons presented in the chapter’s introduction. We can circumvent these issues by computing an overapproximation $G' \supseteq \text{BackReach}_{\leq \Delta}(\text{BackReach}(U, s_c), cc')$ based on an abstract model cc' of the complex controller. For example, the actual outputs generated by cc can be abstracted by the range of values that are valid outputs for the actuators. Since $G'$ overapproximates $G$, using $G'$ as the switching set also guarantees overall system safety (but may introduce pessimism based on the accuracy of the overapproximation). The next sections will describe methods of computing overapproximations of reachability and time-bounded reachability.

4.2.2 Quantization-based Approach for Computing Reachability

We will now a quantization-based approach for computing overapproximations of reachability and time-bounded reachability. Although this approach uses discrete abstractions, it is different from discrete abstraction algorithms discussed in earlier work [104, 105]. Existing approaches consider which neighboring regions can be reached from each region, and then perform a fixed-point computation to determine reachability. The approach presented here, on the other hand, will construct a discrete abstraction where every evolution of the system of up to some $\delta$ time will have a corresponding discrete transition. Therefore, by looking at all outgoing edges from a discrete state, $\delta$-reachability can be overapproximated (which is necessary for computing the Simplex decision logic in Equation 4.1 in Section 4.2.1).

In this section, we will first present a formalization of hybrid systems and their discrete abstractions in Section 4.2.2.1. Next, Section 4.2.2.2 uses the developed formalism to describe a quantization-based approach which can be used to overapproximate reachability and time-bounded reachability. Finally, we describe using the quantization approach on an off-road vehicle rollover case study in Section 4.2.2.3.

4.2.2.1 Hybrid Systems Formalism

Let $V$ be a set of variables. Each variable $v \in V$ is associated with a type which defines the set of values $v$ can take. The set of valuations of $V$ is denoted by $\text{val}(V)$. A variable may be discrete or continuous. Typically, discrete variables model protocol or software state, and continuous variables model physical quantities such as time, position, and velocity. For a subset $S \subseteq \mathbb{R}^n$, we denote by $S_i$ the projection of $S$ on the $i^{th}$ coordinate, $0 \leq i \leq n$. 
Hybrid Automata and Discrete Abstractions

Discrete abstractions for cyber-physical systems are mathematically modeled as automata or labeled transition systems.

**Definition 1.** A Labeled Transition System (LTS) $A$ is a tuple $(Y, \Theta, A, D)$ where

1. $Y$ is a set of variables; elements of $\text{val}(Y)$ are called states,
2. $\Theta \subseteq \text{val}(Y)$ is a set of initial states,
3. $A$ is a set of actions, and
4. $D \subseteq \text{val}(Y) \times A \times \text{val}(Y)$, is a set of transitions. For an element $(y, a, y') \in D$, we write $y \xrightarrow{a} y'$.

An execution of LTS $A$ is an (possibly infinite) alternating sequence $y_0a_1y_1a_2\ldots$, where $y_0 \in \Theta$, and for each $i$, $y_i \xrightarrow{a_i} y_{i+1}$. The set of executions and reachable states of $A$ are denoted by $\text{Exec}_A$ and $\text{Reach}_A$. Given a set of states $S \subseteq \text{val}(Y)$, the set $\text{Prev}_A(S)$ is defined as the set of states from which some state in $S$ can be reached in a single transition of $A$, that is, $\{y \in \text{val}(Y) \mid \exists y' \in S, a \in A, y \xrightarrow{a} y'\}$. $\text{BackReach}_A(S)$ is the least fixpoint of the $\text{Prev}_A$ starting from $S$.

Here, we work with input/output-free hybrid automaton models for cyber-physical systems. A Hybrid Automaton (HA) is a non-deterministic state machine whose variables may change, (a) instantaneously through transitions, or (b) continuously over an interval of time following a trajectory. A trajectory for a set of variables $V$ models continuous evolution of the values of the variables over an interval of time. Formally, a trajectory $\tau$ is a map from a left-closed interval of $\mathbb{R}_{\geq 0}$ with left endpoint 0 to $\text{val}(V)$. The first state of $\tau$, $\tau.\text{fstate}$, is $\tau(0)$. A trajectory $\tau$ is closed if the domain of $\tau$ is $[0, t]$ for some $t \in \mathbb{R}_{\geq 0}$, and we define the last state in $\tau$, $\tau.\text{lstate}$, as $\tau(0)$.

**Definition 2.** A Hybrid Automaton (HA) $A$ is a tuple $(V, \Theta, A, D, T)$ where

1. $V$ is a set of variables; the elements of $\text{val}(V)$ are called states,
2. $\Theta \subseteq \text{val}(V)$ is the set of start states.
3. $A$ is a set of actions.
4. $D \subseteq \text{val}(V) \times A \times \text{val}(V)$ is a set of transitions. A transition $(v, a, v') \in D$ is written in short as $v \xrightarrow{a} A v'$ or as $v \xrightarrow{a} v'$ when $A$ is clear from the context.
5. $T$ is set of trajectories for $V$ that is closed under prefix, suffix, and concatenation \cite{9}. In addition, $A$ is non-blocking, that is, at any state $v \in \text{val}(V)$, either an action can occur or time can elapse.

An execution fragment of $A$ is a finite or infinite sequence $\tau_0 a_1 \tau_1 a_2 \ldots$, such that for all $i$ in the sequence, $\tau_i.\text{lstate} \xrightarrow{a_i+1} \tau_{i+1}.\text{fstate}$. An execution fragment is an execution if $\tau_0.\text{fstate} \in \Theta$. The first state of an execution fragment $\alpha$, $\alpha.\text{fstate}$, is $\tau_0.\text{fstate}$, and for a closed $\alpha$, its last state, $\alpha.\text{lstate}$, is the last state of its last trajectory. The set of executions and reachable states of $A$ are denoted by $\text{Exec}_A$ and $\text{Reach}_A$.

**Definition 3.** Given a hybrid automaton $A = \langle V, \Theta, A, D, T \rangle$, a set of variables $Y$, and a mapping $f : \text{val}(V) \rightarrow \text{val}(Y)$, a Discrete Abstraction of $A$ is a labeled transition system $B = \langle Y, \Theta', A \cup \{\text{Time}\}, D' \rangle$, where

1. $y \in \Theta'$ iff $\exists v \in \Theta, f(v) = y$,

2. $(y, a, y') \in D'$ iff

   (a) transition $a \in A$ and $\exists (v, a, v') \in D$ such that $f(v) = y$ and $f(v') = y'$, or

   (b) trajectory $a = \text{Time}$ and $\exists \tau \in T$, such that $\tau$ is closed and $f(\tau.\text{fstate}) = y$ and $f(\tau.\text{lstate}) = y'$.

In other words, $B$ is a time abstract transition system that simulates $A$; every discrete transition $v \xrightarrow{a}_A v'$ of $A$ is simulated by a discrete transition $f(v) \xrightarrow{a}_B f(v')$ of $B$, and every closed trajectory $\tau$ of $A$ is simulated by $f(\tau.\text{fstate}) \xrightarrow{\text{Time}}_B f(\tau.\text{lstate})$ of $B$.

**Formal Hybrid Automaton Model of Simplex**

In constructing hybrid automata models for Simplex-based embedded control systems, we consider automata in which an opportunity for high-level control mode switches arise once every $\Delta$ unit of time. Such hybrid automata often arise in modeling periodically scheduled real-time and embedded systems \cite{106}. In this section, we define a class of hybrid automaton with a mode variable, which is suitable for capturing Simplex-based control systems. The possible values of mode correspond to the active controller, either the safety controller or the complex controller.

Automaton Simplex, shown in Figure 4.4, captures an abstract $n$-dimensional Simplex-based control system. The Simplex specification is parameterized by a positive constant $\Delta$. The three
variables are (a) continuous state variables $X = \{x_1, \ldots, x_n\}$ each of type $\mathbb{R}$, (b) discrete state variable $\text{mode}$ which can take two values, namely \{safety, complex\}, initialized to safety, and (c) clock variable $c$, initialized to 0. We denote the vector of $n$ continuous variables as a single $x$. We say that the system is in the safety (complex) mode when $\text{mode} = \text{safety}$ (and complex, respectively).

When the system is in the safety (complex) mode, the continuous variables evolve according to the differential equations $\dot{x} = F_S(x) ; \dot{c} = 1$ ($\dot{x} = F_C(x) ; \dot{c} = 1$), where the right hand side of the differential equations $F_S$ ($F_C$) are real-valued continuous functions, and subject to restrictions that we outline later in Section 4.2.2.2. Furthermore, the length of all trajectories of Simplex are upperbounded by $\Delta$. Every $\Delta$ time, exactly one action occurs, $\text{sw}_s$ or $\text{sw}_c$. As a result, the corresponding discrete transition Simplex may or may not change the value of $\text{mode}$; the value of $x$ does not change. The value of $\text{mode}$ changes from safety to complex if, at the pre-state of the transition, the valuation of $x$ is in $G_c$. Similarly, $\text{mode}$ changes from complex to safety if $x$ is in $G_s$ in the pre-state.

Thus, given the dynamics of the system ($F_s$ and $F_c$) under the safe and the complex controllers, and a safety property specified by an unsafe set $U$, the task of designing the decision module for a Simplex-based control system boils down to finding $G_s$ and $G_c$ such that (a) (safety) $\text{Reach}_{\text{Simplex}} \cap U = \emptyset$, and (utility) as time goes to infinity, the duration of time for which $\text{mode} = \text{complex}$ also goes to infinity. Here, we will only focus on formally proving the safety property.

For the simplicity of exposition, we have described Simplex with two modes, each of which has continuous dynamics described by the differential equations $F_S$ or $F_C$. The design methodology described in the next section, however, generalizes to systems where the safety and the complex mode dynamics are described their own hybrid automaton (under some restrictions, described below), with several (sub-)modes with their own discrete switches.
4.2.2.2 Quantization-based Discrete Abstractions

In this section, we will first discuss the restrictions we place on the hybrid automata associated with each mode. Next, we use a quantization-based abstraction function to algorithmically construct discrete abstractions of the hybrid automaton for each of the individual controller modes of the aggregate Simplex hybrid automata. These automata are, in turn, used to verifiably determine the mode guards, $G_c$ and $G_s$ for the combined Simplex system.

Recall, the set of variables for Simplex are $V = \{x, mode, c\}$, where $x$ is a continuous variable of type $\mathbb{R}^n$, $mode$ is a \{safety, complex\}-valued discrete variable, and $c$ is a real-valued clock. We define $\text{Simplex}_{\text{safety}}$ and $\text{Simplex}_{\text{complex}}$ to be the hybrid automata corresponding to the safety and complex modes of Simplex. That is, $\text{Simplex}_{\text{safety}}$ and $\text{Simplex}_{\text{complex}}$ are each a hybrid automaton with variables $\{x_1, \ldots, x_n, c\}$. For a state $x \in \text{val}(\{x_1, \ldots, x_n, c\})$, we refer to the values of the $n$ continuous components by $x.x_1, \ldots, x.x_n$, or aggregates as $x.x$, and the value of the clock as $x.c$.

The unsafe states for the system are specified by a predicate $U_x$ on the continuous state space, $\mathbb{R}^n$, and thus, the overall unsafe set for either $\text{Simplex}_{\text{safety}}$ or $\text{Simplex}_{\text{complex}}$ is $U = U_x \times \mathbb{R}_{\geq 0}$.

We construct the discrete abstractions $\text{AbsSimplex}_{\text{safety}}$ and $\text{AbsSimplex}_{\text{complex}}$ of the given concrete $\text{Simplex}_{\text{safety}}$ and $\text{Simplex}_{\text{complex}}$ automata by quantizing the continuous state space. The construction technique for the $\text{AbsSimplex}_{\text{safety}}$ and $\text{AbsSimplex}_{\text{complex}}$ abstract discrete transition systems is analogous. Therefore, we now only refer to the construction of an $\text{AbsSimplex}_{\text{mode}}$ labeled transition system which abstracts a $\text{Simplex}_{\text{mode}}$ automaton, and assume the corresponding differential equations, referred to simply as $F$, are being used for the mode under consideration.

There are two types of discrete transitions in a discrete abstraction, those arising from transitions in the hybrid automaton (from Definition 3.a in Section 4.2.2.1), and those arising from trajectories (from Definition 3.B in Section 4.2.2.1). In our abstraction procedure, we will consider a restricted class of hybrid automata which have disjoint mode invariants, and do not contain resets for the continuous variables. In this type of automaton, the evolution of the continuous variables does not contain discontinuities, even between discrete mode transitions. Furthermore, given a valuation of the continuous variables, the discrete mode of the hybrid automaton can be uniquely determined. In the constructed discrete abstraction, we will have the unique property that each discrete transition corresponds to an evaluation of up to $\Delta$ time (for some constant $\Delta$ used in the construction) in the corresponding hybrid automata. We will use this property later to construct the Simplex decision module behavior using the discrete abstraction. Each discrete transition in the abstraction, therefore, can correspond to a series of discrete transitions and trajectories in the hybrid automata. The
notions of reachability and backreachability can still be defined using fixpoints as in Section 4.2.2.1.

Dynamics Restrictions

In the algorithm presented here, we place three restrictions on the mode’s differential equations, $F$, which, in turn, restrict the underlying dynamics. Systems with dynamics that do not meet these restrictions are not applicable with this approach (we will later present algorithms which relax some of these restrictions). Later, in our case study in Section 4.2.2.3, we show that these restrictions are not overbearing by applying the developed technique towards rollover prevention of autonomous off-road vehicles.

We first list the three restrictions on the differential equations, $F$, and then describe each one in more detail.

**R1.** For each continuous variable $x_i$, there exist upper and lower bounds $x_{il}$ and $x_{iu}$, such that for all reachable states $v$ of $\text{Simplex}_{\text{mode}}$, $v.x_i \in [x_{il}, x_{iu}]$.

**R2.** For any continuous variable $x_i$, the right hand side of the (possibly nondeterministic) differential equation for $\dot{x}_i$ is expressible as a (possibly nondeterministic) function $f_i(v.x)$. We assume that, for any product of half-open intervals $S \subseteq \mathbb{R}^n$, there exist computable bounds, $l_S, u_S \in \mathbb{R}$, on the derivatives such that, for all $v.x \in S$, $f_i(v.x) \in [l_S, u_S]$.

**R3.** The dependency graph of the continuous variables induced by the differential equations $\dot{x} = F(x)$ is acyclic, save for self-dependencies.

Many physical variables such as velocity, acceleration and temperature, can be associated with upper and lower bound values in the context of a given system. Such bounds can be used in place of $x_{il}$ and $x_{iu}$ in Restriction R1. This implies that our restricted class of $\text{Simplex}_{\text{mode}}$ models can not capture, for example, an unbounded position of a particle on the real line. This requirement is necessary for the constructed abstract discrete transition system to be of finite size, and therefore directly usable by a model checker to determine the Simplex guards. The second restriction, R2, requires that, for every trajectory $\tau$ in $\text{Simplex}_{\text{mode}}$ that is within an arbitrary product of half-open intervals $S \subseteq \mathbb{R}^n$, it is feasible to determine bounds on the minimum and maximum rate of change (differential inclusions) of any continuous variable along $\tau$. The simplest example of this is if the
minimum and maximum possible rates of change of a variable remain constant with respect to time over all \( \tau \). For example, the rate of change of the angular velocity of a wheel is always bounded by the minimum and maximum angular acceleration that can be produced by the associated motor over all operating conditions (assuming there are no brakes). Such derivative bounds can also be derived for variables which change based on their own values, or the values of other variables. This would be the case if, for example, the rate of change of the angular velocity of a wheel was modeled as depending on the current angular velocity of the wheel. As we approach motor saturation (a high angular velocity), we can no longer increase the angular velocity as rapidly as we could when at rest. An example of a dependency on another variable would be if the rate of change of the angular velocity was modeled based on the pitch angle of the road on which we were driving. Symbolically, suppose \( x_1 \) evolves according to \( \dot{x}_1 = f_1(x_1, x_2) \). Given a closed interval \( I_1 \) of values of \( x_1 \) and a closed interval \( I_2 \) for values of \( x_2 \), \( R_2 \) requires that we can compute \( u, l \) such that for all states \( v \) where \( v.x_1 \in I_1 \) and \( v.x_2 \in I_2 \), \( f_1(v.x_1, v.x_2) \in [l, u] \). This restriction is necessary to be able to apply our algorithm for constructing the transitions of the abstract discrete transition system.

Restriction \( R_2 \) also limits the sorts of systems for which we can form discrete abstractions without introducing significant pessimism. Particularly, if we can determine the exact discrete state of the hybrid automaton \( \text{Simplex}_{\text{mode}} \) which the system is in from only the values of \( v.x \), finding such bounds does not add significant pessimism into the abstract system we are constructing (the bounds from the states corresponding to the product of half-open intervals under consideration can be determined directly). However, if the hybrid automaton \( \text{Simplex}_{\text{mode}} \) can be in multiple discrete states for a particular value of \( v.x \), the bounds must encompass bounds from all possible discrete states, which may introduce pessimism depending on the variation in the differential equations at the different states. Since we limited the types of hybrid automata we considered to ones where a valuation of the variables can identify the unique mode the hybrid automata is in, for small half-open intervals (sets of valuations close to each other), the pessimism from this approach is small. The effect of this pessimism will be an abstract discrete transition system which may contain spurious transitions which do not reflect actual system dynamics. Since our guards will be determined based on the constructed discrete transition system, we may switch to the safety controller earlier than is necessary because of these spurious transitions.

Restriction \( R_3 \) forbids certain types of feedback interconnections in the underlying system dynamics. Particularly, when the change for each variable \( x_i \) over the change in time is defined as \( \dot{x}_i = f_i(x_i^a, x_i^b, \ldots) \), we obtain a derivative dependency graph. Restriction \( R_3 \) forbids this de-
pendency graph from having circular dependencies, except for self-dependencies. This means, for example, for two variables \( x_1 \) and \( x_2 \), dynamics like

\[
\begin{align*}
\dot{x}_1 &= f_1(x_2) = x_2 \\
\dot{x}_2 &= f_2(x_1) = -x_1
\end{align*}
\]

are not allowed, but dynamics like

\[
\begin{align*}
\dot{x}_1 &= f_1(x_1, x_2) = x_1 + x_2 \\
\dot{x}_2 &= f_2(x_2) = -x_2
\end{align*}
\]

are permitted. This is necessary to be able to apply our algorithm for constructing the transitions of the abstract discrete transition system. This restriction will be relaxed in later algorithms in this chapter.

**Abstraction Function**

When defining our abstraction function, which takes a concrete state in \( \text{Simplex}_{\text{mode}} \) and maps it to an abstract state in \( \text{AbsSimplex}_{\text{mode}} \), we first fix a collection of quantization parameters \( q_1, \ldots, q_n \in \mathbb{R}_{>0} \), one for each continuous variable. The choice of quantization parameters is a tunable trade off, where larger constants will lead to more pessimism in the associated discrete transition system (which means the decision module derived based on the discrete transition systems will be more pessimistic, possibly switching to the safety controller before it is absolutely necessary), and smaller constants lead to discrete transition systems with more discrete states (which increases the amount of time necessary for a model checker to iterate the system to determine the decision module’s switching guards).

A \( \text{Simplex}_{\text{mode}} \) discrete transition system has set of variables \( Y = \{y_1, \ldots, y_n\} \), where each \( y_i \) is integer-valued. For \( y \in \text{val}(Y) \), we refer to values of the \( n \) components by \( y.y_1, \ldots, y.y_n \).

The quantization-based abstraction function, \( \text{quantize} : \mathbb{R}^n \rightarrow \text{val}(Y) \), is defined as: for any \( x \in \mathbb{R}^n \), \( \text{quantize}(x) = y \), where for each \( y_i \in Y \), \( y.y_i = \lfloor \frac{y_i}{q_i} \rfloor \). Notice that the explicit notion of clock is dropped when performing the abstract mapping. Instead, each transition in \( \text{AbsSimplex}_{\text{mode}} \) corresponds to a series of trajectories and/or discrete transitions in the \( \text{Simplex}_{\text{mode}} \) automaton, with total duration at most \( \Delta \) (where \( \Delta \) is, again, the Simplex controller iteration time parameter). The abstraction function defines the states and transitions of \( \text{AbsSimplex}_{\text{mode}} \) according
to Definition 3. An algorithmic way to construct these transitions will be discussed below. For convenience, we also define two related functions, QUANTIZE-SET and QUANTIZE\(^{-1}\). The function QUANTIZE-SET : \(\mathbb{R}^n \rightarrow 2^{\text{val}(Y)}\) is the extension of QUANTIZE to larger sets of \(\mathbb{R}^n\), while QUANTIZE\(^{-1}\) is the concretization map, QUANTIZE\(^{-1} : \text{val}(Y) \rightarrow 2^{\mathbb{R}^n}\), which is the inverse of QUANTIZE. The concretization of an abstract state gives us the corresponding product of half-open intervals in the continuous space, specifically,

\[
\text{QUANTIZE}^{-1}(y) = \prod_{i=1}^{n} \left[ q_i y_i q_i^* (1 + y_i y_i) \right].
\]

Algorithm for Constructing Discrete Transitions

Assumption R1 implies that for each continuous variable \(x_i\), we can restrict the type of \(x_i\) to the closed interval \([x_{il}, x_{iu}]\) of \(\mathbb{R}\) without altering the behavior of the Simplex\(_{\text{mode}}\) automaton model.

Since each quantization parameter \(q_i\) for the corresponding variable \(x_i\) is strictly positive, it follows that the type (set of all possible values) of the abstract variable \(y_i\) is the finite set \(\{\lfloor \frac{x_{il}}{q_i} \rfloor, \ldots, \lfloor \frac{x_{iu}}{q_i} \rfloor\}\).

Since there are a finite number of abstract discrete states for each variable, and there are a finite number of variables, there are a finite number of abstract discrete states in the aggregate system consisting of all the variables. Therefore, we construct the possible transitions of the abstract system by considering one abstract state \(y\), and repeat the procedure over the finite number of abstract system states.

Requirement R3 rules out circular dependencies (except self-dependencies) in defining the derivatives of continuous variables within a trajectory. Therefore, ignoring self-dependencies, we can form a directed acyclic graph (DAG) of these dependencies and perform a topological sort. For the rest of the algorithm, we assume the variables, \(x_1, x_2, \ldots, x_n\), are ordered according to this topological sort.

We now determine abstract transitions which correspond to trajectories of Simplex\(_{\text{mode}}\) from a particular abstract state \(y\). We first determine the reachable abstract values for the \(y_1\) variable, then the reachable abstract values for the \(y_2\) variable, and so on. After this, we combine the cross product of the reachable values of all the variables to construct the possible transitions from state \(y\). The new goal, then, is determining the reachable abstract values for each variable from state \(y\).

Initially, we consider, \(x_1\), which is has no non-self incoming edges in the derivative dependency DAG (since the variables are ordered according to the DAG’s topological sort). Since \(x_1\) has no non-self incoming edges, the bound function for its derivative, \(\dot{x}_1\), is defined as either a constant, or a function of \(x_1\). In both cases, by requirement R2, it is feasible to determine bounds on the minimum
Figure 4.5: A state (grey) interval within a self-dependent variable is evaluated for abstract discrete reachability within one time step. The updated product of half-open intervals in the associated dimension would reflect the concrete values corresponding to the reachable intervals, in this case [0,4).

and maximum rate of change of $x_1$, the continuous variable, by considering a half-open interval corresponding to the current possible concrete states. This interval can be obtained by considering the dimension corresponding to $y_1$ of the current possible concrete states, $\text{QUANTIZE}^{-1}(y)_1$.

However, since the bounds on $\dot{x}_1$ are only valid within the interval, the bounds may change if the variable enters another interval before $\Delta$ time passes (recall that we are trying to determine the reachable values of the variable in up to $\Delta$ time in the construction). Consider the situation shown in Figure 4.5. Here, an initial state corresponding to the grey interval is evaluated for reachability. If we were to only consider the derivative bounds at the current interval, $\dot{x} = [-0.5, 0.5]$, we would not consider the state corresponding to the interval [3, 4) as reachable in one time step, whereas in actuality if the value of the variable enters the state corresponding to the interval [2, 3), our maximum derivative increases and we may, in fact, reach the interval [2, 3).

In general, to determine the minimum-valued abstract state we can reach in up to $\Delta$ time, we start at the lower bound of the current state’s interval and iterate the discrete states using the minimum derivative until we exhaust $\Delta$ time (or the minimum derivative becomes nonnegative), using new minimum derivatives as we enter new intervals. To determine the maximum state we can reach after up to $\Delta$ time, we start at the upper bound of the current state’s interval and iterate the discrete states using the maximum derivative until we exhaust $\Delta$ time (or the maximum derivative becomes nonpositive), again, using new maximum derivatives as we enter new intervals. The reachable values of the variable under consideration in up to $\Delta$ time, then, are bounded between the two resultant states. We therefore have determined the reachable values for this variable in up to $\Delta$ time, and can add the corresponding discrete transitions into the discrete abstraction we are constructing.

After determining the reachable values for the first variable in the topological sorted order, we update the product of half-open intervals for computing derivative bounds to reflect the values of
Construct Transitions[]
{
  for each \( y \in \text{val}(Y) \)
    \( S \leftarrow \text{quantize}^{-1}(y) \)

  for each \( y_i \in y \) in the topologically sorted order
    reach_{min} \leftarrow \text{MinReach}(\Delta, S, qy_i, i)
    reach_{max} \leftarrow \text{MaxReach}(\Delta, S, qy_i \times (1 + y_{yi}), i)
    S \leftarrow S_1 \times \ldots \times S_{i-1} \times \left[ \text{reach}_{min}, \text{reach}_{max} \right] \times S_{i+1} \times \ldots \times S_n

  for each \( y' \in \text{quantize-set}(S) \)
    \( D \leftarrow D \cup \{(y, \text{Time}, y')\} \)
}

Value MinReach(Time \( t \), Subset \( S \), Value \( x \), DimensionNumber \( i \))
{
  \( S \leftarrow S_1 \times \ldots \times S_{i-1} \times [q_i \times [x/q_i], q_i \times (1 + [x/q_i])] \times \ldots \times S_n \)
  \([l, u] \leftarrow \text{DerivativeBounds}(i, S)\)

  if \( t \geq 0 \) // derivative is positive, done
    return \( q_i \times [x/q_i] \)
  else if \( t + l \times q_i < 0 \) // time expires, done
    return \( q_i \times [x/q_i] \)
  else // otherwise consider next interval
    return \( \text{MinReach}(t + l \times q_i, S, q_i \times ([x/q_i] - 1), i) \)
}

\( x_1 \) reachable in up to \( \Delta \) time. Thus, when computing the derivative bounds of any variables whose derivatives depend on the value of \( x_1 \), all values of \( x_1 \) reachable in up to \( \Delta \) time will be considered.

The algorithm then continues iterating each variable in the topologically sorted order. Thus, for evaluating the reachable values for an arbitrary variable \( y_j \), we will always have the intervals of reachable values in up to \( \Delta \) time for every dependent variable, except possibly \( y_j \). We can then determine the reachable values of variable \( y_j \) by using the iterative procedure outlined above, starting with the variable interval corresponding to \( y_j, \text{quantize}^{-1}(y_j) \).

After we determine the abstract states reachable for each variable starting from \( y \), the cross product of the reachable values for each variable encodes the possible discrete transitions corresponding to all possible continuous evolution of the variables in up to \( \Delta \) time. If we repeat this procedure for every discrete state in the system, we will get a complete labeled transition system, \text{AbsSimplex}_{\text{mode}}\), which is an abstraction of the continuous automaton, \text{Simplex}_{\text{mode}}\).

The complete algorithm is outlined below. The algorithm uses the MinReach function (and MaxReach, which is similar) which is also defined. Here, \( D \) refers to the set of discrete transitions.

Prior to the start of the algorithm, the variables are ordered according to the topological sort of the graph formed by their differential equation dependencies, ignoring self-dependencies.

Here, DerivativeBounds is the function which returns the differential inclusions, mandated by
Requirement R2. Notice that, depending on the Simplex time parameter, $\Delta$, and the quantum parameters for each variable, $q_1, q_2, \ldots, q_n$, the result of this process may have some spurious abstract transitions which are not in the concrete, continuous system. However, it is still a simulation of the concrete system, in that any concrete combination of trajectories and discrete changes in the concrete system, $\text{Simplex}_{\text{mode}}$, will have a corresponding set of transitions in the abstract system, $\text{AbsSimplex}_{\text{mode}}$. By applying this technique for the two Simplex modes, you obtain two discrete transition systems which are discrete abstracts of their corresponding mode.

Finding Guards from Discrete Abstractions

Having constructed the discrete abstractions for the individual modes of Simplex, $\text{AbsSimplex}_{\text{safety}} \triangleq S$ and $\text{AbsSimplex}_{\text{complex}} \triangleq C$, we now describe the verifiable construction of the guards $G_c$ and $G_s$ that constitute the switching logic in the Simplex decision module in Figure 4.1. Suppose that the unsafe set of states is $U$. A safe switching predicate $G_c$ can be computed as:

$$G_c = \text{QUANTIZE}^{-1}(\text{Prev}_C(\text{BackReach}_S(U))).$$

In other words, we consider all the set $B$ of states that are backwards reachable from $U$ using the discretized safety controller. Then we consider the set $C$ of states that can reach $B$ using a single step in the complex controller discrete abstraction (which corresponds to up to $\Delta$ time elapsing in the concrete system) and set $G_c$ as the set of concrete states corresponding to $C$. At a high level, states in $G_c$ are ones where, if we use the complex controller for one control iteration (for $\Delta$ time) and then switch to the safety controller, the system state may still eventually enter the unsafe region. Therefore, during execution, if the current state is in $G_c$, we must use the safety controller immediately. By starting in an initial state that is not in $G_c$ and following this strategy, the system model will never enter an unsafe state.

In the next section, we illustrate using a model checker to perform this procedure within an case study.

4.2.2.3 Off-Road Vehicle Rollover Case Study

We now describe a case study which aims to formally provide property guarantees for autonomous off-road vehicles. Although we simultaneously investigated several integrity and performance properties [107], here we discuss only off-road vehicle rollover prevention. After a brief problem description, we describe the discrete model generation, which is based on the theory developed earlier.
in Section 4.2.2.2. Then, we use a model checker to determine the Simplex automaton guards, as well as verify the correctness of the aggregate system. Finally, we discuss creating a verified and precisely-timed implementation of the decision module based on these guards.

Throughout the case study, we work with the Maude [108] model checker. Maude is a high-performance rewriting framework which uses rewrite equations and rewrite rules to codify models, both of which can be specified conditionally. While rewrite equations perform serial and deterministic computations, rewrite rules can be used to encode parallelism and nondeterminism. Since our system derivatives are bounded by closed intervals, nondeterministic rewrite rules are used to discretely encode the nondeterministic system dynamics. The Maude engine can then exhaustively model check and verify properties about all possible executions of the system.

**Off-road Vehicle Rollover Description**

Although there is no operator inside to injure, autonomous vehicle rollovers may cause damage to vehicles and damage to property. The high-level system may be designed to avoid high-risk situations such as high-sloping terrain, sharp turning, and high speeds, but since automatic controllers on off-road vehicles run sophisticated algorithms, integrating proprietary with commercially-available software, often using COTS hardware, formally verifying rollover avoidance is difficult. A Simplex implementation for rollover prevention is therefore a simpler system to validate.

We must describe the rollover property formally in order to verify that a model of an autonomous off-road vehicle never experiences rollover. We pessimistically state that the rollover property is violated whenever one of the wheels of the vehicle lifts off the ground, which is never part of normal off-road vehicle operation. The relevant physical constants and variables are shown in Figure 4.6. The equation to determine if the rollover property is violated is:

\[
g < \frac{v^2 \sin(\beta)}{\text{WB}} \sin(\tan^{-1}\left(\frac{2H}{\text{TR}}\right)) \cos(\tan^{-1}\left(\frac{2H}{\text{TR}}\right) + \alpha)
\]

\(g\): gravitational acceleration  
\(v\): vehicle velocity  
\(\beta\): vehicle steering angle  
\(\alpha\): slope terrain angle  
\(\text{WB}\): vehicle wheel base  
\(2H\): twice the vehicle’s height of center of gravity  
\(\text{TR}\): vehicle track width
The rollover condition is violated whenever $F_{\text{lift}}$ exceeds $F_{\text{stability}}$. Here, $\omega$ is the angular velocity, calculated as $v \cdot \sin(\beta)/\text{WB}$. The angle $\theta$ is computed as $\tan^{-1}(2H/TR)$.

There is also a symmetric rollover case where $\alpha < 0$ and $\beta < 0$. Additionally, from this equation we can infer the behavior of the safety controller. To avoid rollover, it is clear from the equation that one should reduce the steering angle $\beta$, and reduce the velocity $v$, which matches our intuition about rollover.

**Discrete Model Generation**

Once we know relevant variables for the property we are interested in verifying, we need to construct the hybrid automaton for the tractor system. The variables relevant to rollover are the angle of the terrain $\alpha$, the steering angle $\beta$, and the velocity of the tractor $v$. Within our case study, we constructed a parameterized model generator, which takes as input about 30 parameters about the off-road vehicle system (such as the absolute maximum and minimum velocity, the discretization quantum for each variable, the rates of change of the steering angle, the track length and the wheel base), and outputs a Maude model which encodes the associated discrete abstraction. Our model consists of three variables, the slope angle $\alpha$, the steering angle $\beta$, and the velocity $v$. The rates of change, $\dot{\alpha}$, $\dot{\beta}$, and $\dot{v}$, depend only on their own values, and $\dot{v}$ has different differential equation ranges depending on the direction of travel, as shown in Figure 4.7. As previously discussed in Section 4.2.2.2 dealing with restriction $R2$, such an automaton for our velocity dynamics can be naturally modeled with our approach by a discrete abstraction, and does not introduce excessive pessimism because of the bounds computation. For each specific value of $v$, there is exactly one corresponding discrete state in the hybrid automaton dynamics description.

Our model generator runs the algorithm described in Section 4.2.2.2 and outputs Maude modules.
encoding a discrete transition system for $\text{AbsSimplex}_{\text{complex}}$, which simulates the possible dynamics for the off-road vehicle system when the complex controller is used.

Within a single execution of the Maude model of the complex controller, a nondeterministic execution takes place since, at every point in time, there are many rewrite rules which can be applied to the system. This is expected, since we do not know which possible action the complex controller will take and there are always several options. More importantly, when the Maude engine performs model checking on the system, it iterates all possible rewrite rules exhaustively, allowing us to verify properties for all possible behaviors of the complex controller. For the safety controller, where the vehicle slows down and reduces the steering angle, the model generator creates rewrite equations describing the maximum rate at which we can slow down and the maximum rate at which we can straighten the wheels, based on the input parameters. The safety controller’s dynamics, and the corresponding discrete transition system $\text{AbsSimplex}_{\text{safety}}$, then, are described by transitions where the slope angle, $\alpha$, is allowed to change nondeterministically, but the velocity, $v$, reduces at the maximum allowed rate and the steering angle, $\beta$, decreases at the maximum allowed rate.

**Constructing Simplex Decision Module Guards**

After we have Maude models corresponding to $\text{AbsSimplex}_{\text{complex}}$ and $\text{AbsSimplex}_{\text{safety}}$, we can use the model checker to determine the guards for our system. We primarily focus on determining the guard $G_c$ from Figure 4.4, which is the set of states where we should switch from the complex controller to the safety controller. The guard for the reverse transition, from the safety controller to the complex controller, can use $G_s = \overline{G_c}$, or, more likely, $G_s \subset \overline{G_c}$, if a hysteresis is desired to prevent frequent controller switching.

To determine this guard set, we use the discrete transition systems for each mode and the model checker to perform state-space searches. We can use the technique from Section 4.2.2.2 and have
the model checker output the states in $G_c$. These are the states where the complex controller cannot be used because, even if the safety controller is used immediately at the next control decision, rollover can still occur. Notice that this set of states is not the same as the set of states which statically violate the rollover condition, as the model checker and constructed discrete transition systems capture any inertia within the model, as well as all possible changes in the unactuated variables (the slope of the terrain, in this instance).

The behavior of the decision module, then, is to take the quantized current state of the system, and check if it belongs to the state set output by the model checker. If it does, the safety controller must be used. If it does not, the complex controller can actuate the system. The last step is to encode the state set $G_c$ output by the model checker into a form that can be checked online, while the system is running.

**Verifiable Implementation**

To provide a verifiable implementation, we manually constructed linear bounds which captured all the states in $G_c$. After we have determined the bounds on the states where the safety controller should be used, we create the implementation of the decision module. To aid in this step, we have created formal semantics for a subset of VHDL (a hardware description language) within Maude [109], particularly enough of the language to be able to perform bounds checking.

The first step is to create the decision logic implementation in the VHDL semantics syntax within Maude. This code is executable within the Maude model checker (because the formal semantics are defined within Maude), but has a direct correspondence with a VHDL module. Using the Maude-based implementation along with the semantics, we can perform a state-space search on the composite system which will now include the discrete mode transitions determined by the guards, verifying that if we use the bounds as the switching strategy, the system will never enter a rollover state.

The second step is to take the Maude-based implementation and translate it a VHDL module. This is a straightforward step since there is a direct correspondence between the Maude-based VHDL code and regular VHDL code. We have created a program which does this translation automatically. The output of the program, then, is a VHDL module which captures the behavior of the decision module which was formally checked by Maude to result in a system where rollover does not occur. The VHDL code can then be put through a hardware synthesis tool and executed on a Field Programmable Gate Array (FPGA), or turned into an Application-Specific Integrated Circuit.
The advantage of using VHDL as a target language is twofold. First, from a real-time perspective, hardware logic is extremely predictable. Our VHDL Maude semantics include a cycle counter which can be used in conjunction with the model checker to determine an upper bound on the number of clock cycles used to perform the decision module logic (by performing a search for a state where the cycle counter is greater than some value and seeing no matching reachable states). The number of cycles, when combined with a clock frequency, gives us a worst-case execution time. The hardware synthesis tools then check that timing constraints of the VHDL module are met (that the hardware logic can actually execute at the desired clock frequency). The second advantage of targeting VHDL is that performing Simplex at a lower level (in dedicated hardware) has previously been shown to be able to capture more potential errors [2]. By using the lower-level, System-Level Simplex Architecture, our design is more robust.

4.2.3 Increased Reachability Accuracy for More General Systems

We will now discuss another approach for computing reachability different from the quantization-based method presented earlier. Here, reachability is computed using a two-step procedure. Roughly, in the first step, we compute the reachable states in up to $\delta$ time from the initial state, which is added to the current reachable set. In the second step, we compute which states are reachable in exactly $\delta$ time. These are the states that are used as the next initial states in the recursive procedure. Since the states reachable in exactly $\delta$ time are a subset of those reached in up to $\delta$ time, our computed set is smaller compared with standard discrete abstraction approaches. The bigger the $\delta$ used, the larger the potential savings. However, since the $\delta$-reachability algorithm is also an overapproximation, using a large $\delta$ can also lead to larger errors, so a balance must be struck between the two potential sources of error.

Throughout this section, we will augment the discussion with a demonstrative Simple-Vehicle System, which is introduced in Section 4.2.3.1. Section 4.2.3.2 presents the assumptions of our backreachability algorithm. Next, Subsection 4.2.3.3 presents the algorithm for computing bounded and unbounded backreach sets. Next, in Section 4.2.3.4, three strategies are proposed which together can bound the error of a $\text{BackReach}_{\leq \Delta}$ computation to an arbitrary constant. A case study based on a waypoint tracking system, which uses the described algorithm, is then given in Section .

Additionally, although we are concerned with backreachability for Simplex, the explanations are easier to understand, and therefore presented, in terms of forward reachability. The two notions can
Unrecoverable States

(a) if \( \frac{v_{\text{max}}^2}{2 \times a_{\text{min}}} < x_{\text{max}} \)

(b) if \( \frac{v_{\text{max}}^2}{2 \times a_{\text{min}}} > x_{\text{max}} \)

Figure 4.8: The gray area indicates \( \text{BackReach}(U, sc) \), which are states where using the safety controller leads to safety violations for the example from Section 4.2.3.1.

be shown to be computationally equivalent.

### 4.2.3.1 Simple-Vehicle System Example

Consider a vehicle which moves along a one-dimensional line, modeled as a point \( x \) on the x-axis which moves according to the input acceleration \( a \) generated by the controller. Physical constraints require that \( a \in [a_{\text{min}}, a_{\text{max}}] \), and the velocity of the vehicle remains in the range \( [v_{\text{min}}, v_{\text{max}}] \). The safety property requires that the point \( x \) remains in the range \( [x_{\text{min}}, x_{\text{max}}] \), where \( x_{\text{min}} < 0 < x_{\text{max}} \).

For simplicity, we assume \( x_{\text{min}} = -x_{\text{max}} \), \( v_{\text{min}} = -v_{\text{max}} \) and \( a_{\text{min}} = -a_{\text{max}} \).

For this system, the safety controller that we consider is a bang-bang controller which outputs the maximal negative acceleration \( a = a_{\text{min}} \) for \( x > 0 \) and outputs \( a = a_{\text{max}} \) if \( x \leq 0 \). Two possible sets of unrecoverable states (depending on the exact parameters of the plant) are shown in Figure 4.8.

Next, we specify the hybrid automata for the complex controller and plant system, as well as the safety controller and plant system, shown in Figure 4.9. The hybrid automaton for the complex controller and plant system, in Figure 4.9a, has three locations (discrete modes). Under unsaturated operation, the location invariant is \( v_{\text{min}} < v < v_{\text{max}} \) and the derivative equations are \( \dot{x} = v \) and \( \dot{v} = [a_{\text{min}}, a_{\text{max}}] \) (\( \dot{v} \) is nondeterministic). There are two other locations corresponding to when the point has reached its minimum and maximum velocity (labeled min_speed and max_speed) where \( \dot{v} \) is restricted to be either nonnegative or nonpositive. The hybrid automaton for the safety controller and plant system, in Figure 4.9b, contains two locations corresponding to the two states of the controller (labeled forward and backward), and two more locations for when the velocity has reached saturation (labeled min_speed and max_speed). When we compute an overapproximation of \( \text{BackReach} \) or \( \text{BackReach}_{\leq \Delta} \), it will be with respect to one of these automata.
Figure 4.9: The hybrid automata describing the Simple-Vehicle System have no transition guard restrictions, so the discrete location switching is done solely based on the invariants.
4.2.3.2 System Assumptions

In order to apply our algorithm, we have two assumptions, which we outline and elaborate on in the context of the Simple-Vehicle System below.

**Assumption 1:** For any rectangular set of states $H \subseteq S$, for any continuous variable $x_i$, there exist functions $db_{x_i}^{\min}$ and $db_{x_i}^{\max}$, that bound the derivative of $x_i$ with respect to time in $H$. That is, $db_{x_i}^{\min}(H) \leq \frac{dx}{dt} \leq db_{x_i}^{\max}(H)$, for every $x_i$.

**Assumption 2:** We make a distinction between two types of derivative dependencies, explicit ones directly extracted from the differential equations in each location of the hybrid automaton (for example, $\dot{w} = v$ would create an directed edge from the node corresponding to $v$ to the node corresponding to $w$), and implicit dependencies which arise because as time advances, the continuous state may cause a change in hybrid-automaton locations which causes the differential equations of variables to be changed. Assumption 2 restricts the systems we consider to those where the *explicit* dependency graph of the state-variable derivatives does not have cycles, except for self-loops.

*Example:* In the context of the Simple-Vehicle System, to meet Assumption 1, we must provide the derivative bounds functions for each variable, $x$ and $v$, which can be automatically extracted from the hybrid automata. These functions take as input a rectangle of the state space defined by upper and lower bounds on each variable, $(v^{\text{lower}}, v^{\text{upper}}) \times (x^{\text{lower}}, x^{\text{upper}})$.

For the safety controller automaton, $db_{x}^{\min} = v^{\text{lower}}$, $db_{x}^{\max} = v^{\text{upper}}$,

$$db_{v}^{\min} = \begin{cases} -5 & \text{if } v^{\text{upper}} > v_{\text{min}} \\ 0 & \text{otherwise} \end{cases}$$

and

$$db_{v}^{\max} = \begin{cases} 5 & \text{if } v^{\text{lower}} < v_{\text{max}} \\ 0 & \text{otherwise} \end{cases}$$

For the complex controller abstraction automaton, again, $db_{x}^{\min} = v^{\text{lower}}$, $db_{x}^{\max} = v^{\text{upper}}$, and

$$db_{v}^{\min} = \begin{cases} -5 & \text{if } x^{\text{upper}} > 0 \land v^{\text{upper}} > v_{\text{min}} \\ 0 & \text{if } x^{\text{upper}} > 0 \land v^{\text{upper}} \leq v_{\text{min}} \\ 0 & \text{if } x^{\text{upper}} \leq 0 \land v^{\text{upper}} \geq v_{\text{max}} \\ 5 & \text{otherwise} \end{cases}$$
and

\[
db_{\max} = \begin{cases} 
5 & \text{if } x_{\text{lower}} \leq 0 \land v_{\text{lower}} < v_{\text{max}} \\
0 & \text{if } x_{\text{upper}} > 0 \land v_{\text{upper}} \leq v_{\text{min}} \\
0 & \text{if } x_{\text{upper}} \leq 0 \land v_{\text{upper}} \geq v_{\text{max}} \\
-5 & \text{otherwise}
\end{cases}
\]

In general, if the dynamics contain nonlinear equations, having the user provide a function to meet Assumption 1 is nontrivial. This essentially because it is a general nonlinear optimization problem. In order to soundly solve this problem, the standard mathematical technique of taking the derivative in each of the dimensions and then checking if the derivative is zero in order to check for minimums and maximums can be applied. In our tool implementation of the described algorithm, in addition to providing a Matlab script to help symbolically compute the function for Assumption 1, we have also added an option to check if this was done incorrectly. If this check is enabled, the tool will sample the derivative function within the rectangular set of states \( H \), and if a derivative lower or higher than what the user has indicated is observed, a warning is given to the user. Of course, sampling is not guaranteed to detect errors if they exist, so this only serves as a way to raise confidence that the function has been provided correctly. Symbolically solving the differential equations for the minimum and maximum is sound and, if it can be done either by hand or using the Matlab script, will provide a correct answer to the global optimization problem.

To meet Assumption 2, we construct and check the derivative dependency graphs (shown in Figure 4.10). For both controllers, in every location, the value of \( \dot{x} \), explicitly depends on \( v \), and the value of \( \dot{v} \) does not explicitly depend on any variables. However, due to the possibility of a change in automaton location, there is an implicit dependence of \( \dot{v} \) on \( v \) in the complex controller abstraction automaton. In the safety controller automaton, there are two implicit dependencies: one implicit dependence of \( \dot{v} \) on \( v \), and a second implicit dependence of \( \dot{v} \) on \( x \). Since both of the explicit dependency graphs (solid arrows) with self-loops removed are acyclic, the system meets the second assumption.
4.2.3.3 Algorithm for Overapproximating Reach and Reach_{\leq \Delta}

In this section, we outline our proposed algorithm for computing overapproximations of Reach and Reach_{\leq \Delta}. We start by presenting the pseudocode of the algorithm, and then elaborate on each of the functions.

The pseudocode uses the following notation: The expression $D.i$ refers to the $i$th element of a finite set $D$ in an arbitrary fixed ordering. The expression $H/_{i}$ refers to the projection of the $i$th dimension of a hyperrectangle $H$. The minimum value in this one-dimensional projection is referred to by $H_{/i}^{\min}$ and the maximum value is referred to by $H_{/i}^{\max}$. The hybrid automaton has $n$ continuous variables, $x_1, x_2, \ldots, x_n$, which are ordered according to the topological sort of the explicit derivative dependency graph with self-loops removed. The values $q_{x_i}$ for each variable $x_i$ are quanta used in the computation which are fixed constants.

In the pseudocode, several functions are also used:

- **getLocations** takes a hyperrectangle state space as input and outputs a set of integers corresponding to the set of locations in which the hybrid automaton may be.

- **getFirstImplicitDerivativeDependency** takes as input an integer corresponding to an automaton location, and returns the implicit derivative dependency of the node corresponding to $x_i$ that comes first in the topologically-sorted variable order.

- **$\alpha$** is an abstraction function which takes as input a set of states, and outputs a set of abstract states represented an integer for each variable. The $\alpha^{-1}$ function is the concretization function which, given an abstract state, returns the set of corresponding concrete states of the system. Formally, if the continuous state space is $\mathcal{X} = \mathbb{R}^n$ and the abstract state space is $\mathcal{Y} = \mathbb{Z}^n$, then $\alpha : \mathcal{X} \rightarrow \mathcal{Y}$. $\alpha(x) = y_1, y_2, \ldots, y_n$, where each $y_i = x_i/q_{x_i}$, and $x_i$ refers to the $i$th coordinate of a state $x \in \mathcal{X}$. In the abstraction function, $q_{x_i}$ are the constant quanta for each variable, as mentioned above. As usual, $\alpha^{-1} : \mathcal{Y} \rightarrow 2^{\mathcal{X}}$. In the pseudocode, we use the natural lifting of these functions from a single input state to a set of input states.

The pseudocode for the reachability and time-bounded reachability algorithms is shown in Figure 4.11.

The top-level of the algorithm, the ComputeReach function (lines 1-22), starts by dividing the state space into hyperrectangles based on a provided constant quantum size for each variable through the abstraction function $\alpha$ (line 4). The fixpoint computation loop for computing reachability occurs
// outputs an overapproximation of reachability from an initial set
ComputeReach(I) {
    declare D := α(I);
    declare C := α⁻¹(D);
    declare m := |D|; // number of abstract states
    declare array[m] E;
    declare C' := ∅;
    for i = 1 to m
        E[i] := α⁻¹(D[i]); // initialize the m exact reach sets
    while (C ≠ C') // loop until fixpoint
        C' := C;
        for (i = 1; i < m; i := i + 1)
            (H, l) := ComputeDeltaReach(E[i]);
            E[i] := l; // update exact reach set
            C := C ∪ H; // accumulate reachability
    return C;
}
// outputs (ReachΔ, Reach=Δ) from an input hyperrectangle
ComputeDeltaReach(H) {
    declare E := H;
    declare L := getLocation(H);
    for (i = 1; i < n; i := i + 1) // loop over every variable
        declare i := MinReach(i, H);
        declare u := MaxReach(i, H);
        L[i] := [l, u]; // update exact delta reach
        H[i] := |min(H[i]^min, i), max(H[i]^max, u)]; // update delta reach
    if (L ≠ getLocations(H)) // need to backtrack
        L := getLocations(H);
        i := getFirstImplicitDerivativeDependency(i) - 1;
    return (H, L)
}
// overapproximates the minimum value of x_i reached after δ time
MinReach(i, H) {
    return MinReachRecursive(H, i, δ, H[i]^min, db[i]^min, true);
}
// overapproximates the minimum value of x_i reached
MinReachRecursive(i, H, time, start, db, isFirstInterval) {
    H[i] := [start - q_s, start]; // consider the current interval
    declare der := db(H);
    declare nextIntervalTime := (der > 0 ? time : -q_s/der);
    if (nextIntervalTime < 0) // switch direction
        if (isFirstInterval) // can not safely reverse direction; be pessimistic
            return start;
        else
            return MaxReachRecursive(i, H, time, start, db, false);
    else if (nextIntervalTime ≥ time) // time expires in this interval
        return start + nextIntervalTime * der;
    else // continue to next interval
        return MinReachRecursive(i, H, time - nextIntervalTime, start - q_s, db, false);
}

Figure 4.11: The two-step algorithm for computing reachability is divided into a top-level (ComputeReach), middle-level (ComputeDeltaReach), and low-level function (MinReach/MaxReach).
on lines 13-19. Starting from the hyperrectangle corresponding to every discrete state which contains an unsafe state, we use the ComputeDeltaReach function to compute both the states reachable in up to \( \delta \) time, \( \text{Reach}_{\leq \Delta} \), and the states reachable in exactly \( \delta \) time, \( \text{Reach}_{= \Delta} \) (line 17). The \( \text{Reach}_{= \Delta} \) result is used as the initial set of states for the next iteration of the loop (line 18), while the \( \text{Reach}_{\leq \Delta} \) result is added to the global reachable set (line 19). The intuition behind the correctness of this function is that any state reachable in up to \( k < \delta \) time will necessarily pass through a state reachable in exactly \( \delta \) time (specifically, after \( \delta - k \) time). Therefore, we need not consider all of \( \text{Reach}_{\leq \Delta} \) as the initial states in the next iteration. In terms of termination, notice that this algorithm will clearly not terminate if the computed reach set is infinite. If termination is desired, one can bound the reachable state space with the hybrid automaton, and assure the the variable derivatives do not asymptotically approach 0 (for example, by adding an \( \epsilon \) amount of overapproximation in the \( \text{db}_{x_i}^{\min} \) and \( \text{db}_{x_i}^{\max} \) functions).

The ComputeDeltaReach function (lines 24-42) is used to overapproximate both \( \text{Reach}_{\leq \Delta} \) and \( \text{Reach}_{= \Delta} \). Starting from an input initial hyperrectangle, the sets of states are computed for each variable in the order of the topological sort of the explicit derivative dependency graph with self-loops removed (lines 30-39). This ensures that when the bounds is being computed for a particular variable, all (non-self) dependent variables already have valid computed \( \text{Reach}_{\leq \Delta} \) bounds, ensuring correctness. After a \( \text{Reach}_{\leq \Delta} \) bound for a variable is computed, this bound is used as input to the derivative bounds function that is used to compute \( \text{Reach}_{\leq \Delta} \) for the subsequent dependent variables (because of the assignment on line 35). The \( \text{Reach}_{= \Delta} \) set is also maintained (line 34). After we have computed \( \text{Reach}_{\leq \Delta} \) for every variable, the cross product is an overapproximation of \( \text{Reach}_{\leq \Delta} \) from the initial state (which is iteratively constructed on line 35).

The proposed algorithm does allow models with cycles caused by implicit derivative dependencies, restricting only that the explicit-only dependency graph of derivatives be acyclic except for self-loops. After we compute the \( \text{Reach}_{\leq \Delta} \) for a variable with an implicit derivative dependency which creates a cycle in the combined explicit / implicit derivative dependency graph, we go back and check if we may have entered a new state of the automaton (line 37). If we may have, the algorithm backtracks and recomputes the reachability of all the variables that could have possibly been affected (line 39). Since there are only a finite number of discrete locations in the hybrid automaton, the backtrack process can only happen a finite number of times, and the algorithm remains terminating.

**Example:** Consider computing \( \text{Reach}_{\leq \Delta} \) in the Simple-Vehicle System with respect to the safety controller / plant automaton (Figure 4.9b), starting from \((x = [-2, -1], v = [4, 5])\). We fix the
Figure 4.12: An estimate of $\text{Reach} \leq \Delta$ in the computed using 10,000 simulations (light gray region) for the Simple-Vehicle System is shown in comparison with the overapproximation computed with proposed $\text{Reach} \leq \Delta$ algorithm (solid gray line). This computation is done with respect to the sc automaton. Here, $\delta$ is 0.5 and the initial state is the black square with the slanted pattern ($x = [-2, -1], v = [4, 5]$).

quanta of both dimensions to be 1, fix $\delta$ to be 0.5 time units, fix $v_{\text{max}}$ to be 10, and fix $a_{\text{min}} = -5$ and $a_{\text{max}} = 5$. The algorithm will compute in the order of the explicit dependency graph, first $v$ then $x$. First the reachability for $v$ is computed using $\text{MinReach}$ and $\text{MaxReach}$. These two functions return 6.5 and 7.5, respectively. Due to the capping of the minimum value (line 35) the $\text{Reach} \leq \Delta$ range is set as $v = [4, 7.5]$. Second, the reachability for $x$ is computed to be $x = [-2, 2.75]$. The algorithm can not terminate here, since the change in $x$ may result in a change in hybrid-automaton location (the condition on line 37 is true). The algorithm goes back and recomputes $\text{Reach} \leq \Delta$ for $v$ to be $v = [1.5, 7.5]$. Next, the reachability of $x$ is recomputed once again to be $x = [-2, 2.75]$. Although there is an implicit dependency, the set of possible discrete locations has not changed since the last iteration (the condition on line 37 is false due to the prior assignment on line 38), and the loop terminates. The result of the algorithm is an overapproximation of the actual $\text{Reach} \leq \Delta$ states, shown in Figure 4.12.

The $\text{MinReach}$ and $\text{MinReachRecursive}$ functions (lines 44-66) (and symmetric $\text{MaxReach}$ and $\text{MaxReachRecursive}$, not shown), overapproximate $\text{Reach} \leq \Delta$ for one variable, under the assumption
that the passed-in hyperrectangle contains valid $\text{Reach}_{\leq \Delta}$ bounds for all dependent dimensions. This is done by starting at the minimum value of the variable in the initial hyperrectangle (line 47), and proceeding at the minimum derivative for $\delta$ time, considering new derivatives as we enter new intervals (line 53). Intuitively, this is correct because the all the bounds of the dependent dimensions is correct are assumption so the call to $db$ (line 54) will yield a correct bound on the current variable’s derivative. If the derivative is nondeterministic, we will still overapproximate $\text{Reach}_{=\Delta}$ by considering the minimum in each interval. Special care is taken if the derivative is zero (line 55), or if the minimum derivative is positive (lines 58-61), which avoids infinite loops caused by Zeno behavior through the isFirstInterval variable.

Example: To help illustrate this algorithm, we compute $\text{Reach}_{\leq \Delta}$ from a state in the Simple-Vehicle System, in the cc’ automaton (Figure 4.9a). Now, we will compute $\text{Reach}_{\leq \Delta}$ from the hyperrectangle $(x = [1, 2], v = [8, 9])$. The topological sort of the self-loop-removed derivative-dependency graph (Figure 4.10a) is $v, x$, so we start with the $v$ dimension. The inner loop of the algorithm, MinReach and MaxReach, will compute the minimum and maximum values of $v$ that can be reached after $\delta$ time. In the functions, the variable time is used to keep track of the time elapsed in terms of the execution of the system when computing these values.

Initially, $time = \delta = 0.5$. To compute the minimum reachable velocity, we start by invoking $db_{v}^{min}$ with the hyperrectangle $(x = [1, 2], v = [7, 8])$, which outputs the derivative bounds -5. At the minimum derivative, -5, the next interval is reached in 0.2 time units. We update $time$ to be $0.5 - 0.2 = 0.3$. The process then repeats for the next interval, $v = [6, 7]$. The minimum derivative for $v$ is again -5, and $time$ is updated to 0.1. On the third iteration, $time$ is less than the time it would take to reach the next interval, and the minimum $\text{Reach}_{\leq \Delta}$ value is computed as $v = 4.5$ (line 63).

To compute the maximum velocity we again, initialize $time$ to be $\delta = 0.5$. First, $db_{v}^{max}$ with the hyperrectangle $(x = [1, 2], v = [9, 10])$ outputs 5 as the maximum derivative for the $v$ variable. The $time$ variable is updated to 0.3. On the second iteration, however, $db_{v}^{max}$ with the hyperrectangle $(x = [1, 2], v = [10, 11])$ outputs 0 as the maximum derivative for the $v$ variable, so the next interval is unreachable in 0.3 time units. The maximum $\text{Reach}_{=\Delta}$ value is therefore $v = 10$ (line 63). The $\text{Reach}_{=\Delta}$ values for $v$ are [5, 10]

Now the ComputeDeltaReach function would proceed to compute the reachable values for the next variable, $x$, using the values [5, 10] for the $v$-dimension of the hyperrectangle when calling MinReach and MaxReach.
Computing the maximum value that can be reached for \( x \), the states of the computation proceed as:

\[
\begin{align*}
(t = 0.5, x = [2, 3], d_{x_{\max}}(x = [2, 3], v = [5, 10]) = 10); \\
(t = 0.4, x = [3, 4], d_{x_{\max}}(x = [3, 4], v = [5, 10]) = 10); \\
(t = 0.3, x = [4, 5], d_{x_{\max}}(x = [4, 5], v = [5, 10]) = 10); \\
(t = 0.2, x = [5, 6], d_{x_{\max}}(x = [5, 6], v = [5, 10]) = 10); \\
(t = 0.1, x = [6, 7], d_{x_{\max}}(x = [6, 7], v = [5, 10]) = 10).
\]

where \( t \) is the value of time. At this point, the next interval can not be entered in 0.1 time, and the maximum \( \text{Reach} \leq \Delta \) value is \( x = 7 \).

Computing the minimum value \( x \) can reach, we start with \( t = 0.5 \) and invoke \( d_{x_{\min}} \) with the hyperrectangle \( (x = [0, 1], v = [5, 10]) \). This returns a minimum derivative in the \( x \) dimension of 5, which is nonnegative, so we switch directions and call \( \text{MaxReachRecursive} \) (line 61), with the \( d_{x_{\min}} \) function as a parameter. This will eventually reach a minimum \( \text{Reach} = \Delta \) value of \( x = 6 \). The \( \text{Reach} = \Delta \) values are \( x = [6, 7] \). This example is shown in Figure 4.13.

4.2.3.4 Accuracy Convergence

One important concern for using the proposed algorithm to compute the Simplex switching set is the accuracy of the \( \text{Reach} \leq \Delta \) overapproximation, which in turn affects the accuracy of the global-reachability overapproximation that it computes. Here, we propose three strategies that can be
used to reduce the error of the proposed algorithm. Then, an important accuracy theorem is stated which allows us, under some reasonable assumptions, to reduce the computed \( \text{Reach} \leq \Delta \) error for each variable to an arbitrarily small constant. Finally, through example, the proposed strategies are shown to, in fact, reduce the computed error.

We propose three strategies to reduce the error of the \( \text{Reach} \leq \Delta \) region from an initial hyper-rectangle, which we call the quantum rule, the refine rule and the split rule. We later show that these, when used in combination, can reduce the error in each variable of the \( \text{Reach} \leq \Delta \) overapproximation to an arbitrary constant.

The **quantum rule** uses the fact that the inner loop (\( \text{MinReach} \) and \( \text{MaxReach} \)) of the algorithm considers intervals one quantum in size. Since no restrictions are placed on the minimum size of the quantum, the quantum rule, which evenly splits the size of this quantum for one variable, is always safe to apply. Intuitively, this helps with accuracy by, at each time, providing a smaller hyperrectangle to the derivative bounds function which allows it to output a more accurate derivative bound.

The **refine rule** is based on the fact that \( \text{Reach} \leq \Delta \) from an initial set, \( w \), is equal to the union of \( \text{Reach} \leq \Delta \) from multiple smaller sets, as long as the union of those smaller sets is equal to the initial set \( w \). This is sound because, formally, if \( w = w_1 \cup w_2 \cup \ldots \cup w_n \), then \( \text{Reach} \leq \Delta (w) = \text{Reach} \leq \Delta (w_1) \cup \text{Reach} \leq \Delta (w_2) \cup \ldots \cup \text{Reach} \leq \Delta (w_n) \). This allows us to refine the initial hyperrectangle into smaller hyperrectangles and still obtain a safe overapproximation by taking the union of the results. The rule itself will be applied to a particular variable, and will split the initial hyperrectangle into equal-sized hyperrectangles along that variable. As with the quantum rule, this intuitively helps with accuracy by providing a smaller hyperrectangle to the derivative bounds function which allows it to output a more accurate derivative bound.

The final rule, the **split rule**, is based on the fact that \( \text{Reach} \leq \Delta \) can be decomposed in a way similar to the way in which we described computing full reachability the \( \text{ComputeReach} \) function. Formally, \( d_1 + d_2 + \ldots + d_n = \delta \) implies that \( \text{Reach} \leq \Delta (w) = \text{Reach} \leq d_1 (w) \cup \text{Reach} \leq d_2 (\text{Reach} \leq d_1 (w)) \cup \text{Reach} \leq d_3 (\text{Reach} \leq d_2 (\text{Reach} \leq d_1 (w))) \cup \ldots \). When we say this rule is applied \( n \) times, we split \( \delta \) with \( n \) equal constants, \( d_1 = d_2 = \ldots = d_n = \frac{\delta}{n} \). Intuitively, this rule will improve accuracy by, when computing the reachability for a particular variable, reducing the variable ranges of the dependent dimensions. This provides the derivative bounds function with a smaller hyperrectangle, which allows it to output a more accurate derivative bound.

The application of these three rules can be used to reduce the pessimism of the error in the
Reach$_{\leq \Delta}$ overapproximation in each discrete mode to an arbitrarily small constant in each dimension. This is reflected in the following statement:

Claim 1: By applying the quantum rule, the refine rule and the split rule finitely many times, the maximum error in each variable $x$ of the computed Reach$_{\leq \Delta}$ overapproximation in a single mode, for a fixed $\delta$ and from an initial hyperrectangle $\alpha$, can be reduced to below an arbitrary positive constant $e_x$.

For this to be true, two assumptions are necessary. First, the derivative bounds function should not itself output errors of the actual derivative bounds, as the computed set relies on the accuracy of this function. Second, the derivative function for each variable $x$, $\dot{x} = f(y_1, y_2, \ldots, y_n)$ should be a Lipschitz continuous function with respect to each input variable, and the derivative value $\dot{x}$ should be bounded. This essentially means that we can bound the rate of change of the derivative $\dot{x}$ (and therefore indirectly the rate of change of the value of $x$) in a finite amount of time.

As an empirical demonstration of Claim 1, we show an example in the Simple-Vehicle System using the safety controller / plant with an initial hyperrectangle of $(x = [-2, -1], v = [4, 5])$ with a value of $\delta = 0.5$. The original computed Reach$_{\leq \Delta}$ result, with no applied accuracy-increasing strategies, was previously shown in Figure 4.12. By applying the accuracy-increasing strategies, we can approach the actual Reach$_{\leq \Delta}$ set with arbitrary precision, as shown in Figure 4.14.

4.2.3.5 Case Study: Waypoint Tracking System

We now discuss the proposed sandboxing approach for an autonomous waypoint tracking system (WTS) with a short case study. This system model is inspired by applications such as automated lawn mowers or skid-steer loaders. The autonomous vehicle is required to follow a (predefined) sequence of waypoints while remaining within a fixed safe distance of the line joining successive waypoints.

The controller software periodically senses the position $(x, y)$, the velocity $v$, and the heading $\theta$, of the vehicle and sets the acceleration ($\dot{v}$) and the steering ($\dot{\theta}$) based on the current waypoint $(x^*, y^*)$ of the system. The vehicle models a skid-steer system which can turn in place, i.e., the heading $\theta$ can change even when the velocity $v$ is 0. The equations of motion for the vehicle’s position are given by the following nonlinear differential equations:

$$\dot{x} = v \cos \theta, \quad \dot{y} = v \sin \theta$$

We assume that there is no information available for the complex controller (CC) we wish to
Figure 4.14: An estimate of $\text{Reach}_{\leq \Delta}$ computed using 10,000 simulations (dark gray region) for the Simple-VehicleSystem is shown in comparison with the overapproximation computed with the $\text{Reach}_{\leq \Delta}$ algorithm (light gray region) with various amounts of accuracy-increasing strategies applied.
sandbox, except that it operates within the physical limits of the actuator. That is, $\dot{v} \in [a_{min}, a_{max}]$ and $\dot{\theta} \in [\phi_{min}, \phi_{max}]$. Recall that the safety requirement is to keep the vehicle within some distance of the line joining the waypoints. Thus, a simple safety controller (SC) strategy is to slow down and stop the vehicle as fast as possible.

The Embedded Safety Critical Programming Environment (ESCAPE) toolkit is a set of tools and design methodology we are developing which uses the Simplex Architecture to generate cyber-physical system sandboxes. ESCAPE consists of two parts, (1) HyLink, and (2) SimplexGen. HyLink is a translation tool which takes as its input a Simulink/Stateflow model and translates the model into an hybrid system intermediate format. SimplexGen is an implementation of the algorithm described in this paper which takes as its input, (1) the safety controller / plant model, (2) the abstract complex controller / plant model, (3) the safety invariant to verify, and (4) computation constants (the size of the quanta, the value of the control interval $\delta$). ESCAPE uses these to generate code implementing behavior of a verified decision module. ESCAPE provides a set of Simulink blocks for defining the safety invariant and computation constants, as shown in Figure 4.15a.

The complex controller and safety controller were modeled as hybrid systems using Mathwork's Simulink environment (Figure 4.15b). HyLink is used to extract the hybrid automata from the Simulink/Stateflow models and create an input format which SimplexGen can use. The SimplexGen tool then uses the algorithm described above to generate the switching set. This switching set can then be automatically encoded into a source code file to be used in the Simplex decision module during operation.

For computing the backreach set, the algorithm requires the derivative bounds for each variable. These bounds are obtained from the translated hybrid automata models of the complex controller and safety controller as follows: given a rectangular set $H$ of states, the derivative bounds of each variable in $H$ are determined from the trajectory definitions (differential equations) of the locations of the hybrid automaton whose invariants intersect $H$. The differential equations for each variable $x_i$ are of the form $\dot{x}_i = f(x)$. We then maximize and minimize the function $f(x)$ over the set $H$.

The size of the discrete state space for this example consisted of 1,536,000 states, and the algorithm ran to termination in about 20 minutes. The switching set generated for the WTS is a four-dimensional set $(x, y, v, \theta)$ which cannot be easily visualized. We can, however, analyze the output set by fixing two of the dimensions (for example, $v$ and $\theta$) and plotting the other two dimensions. Figure 4.16 shows two plots of the switching set for two values of $v$ and a fixed $\theta$. By changing the values of $v$ and $\theta$, we can verify our intuition about the switching set: that going away
Custom ESCAPE blocks were created in Simulink to provide the non-model input for SimplexGen. Simulink blocks are used to specify the system dynamics.

Figure 4.15: The ESCAPE toolkit uses Simulink/Stateflow as a front end.

from the waypoint-connecting line segment at high velocity will more quickly switch to the safety controller.

4.3 Related Work

System-Level Simplex

Previous research has been performed on reliable system design. One method proposed to accomplish this has been N-version programming [110]. In this method, multiple versions of software are independently created from the same specification. Then, all are run and the result given by the majority of versions is taken as the output of the system. One drawback with this method is the lack of statistical independence of bugs [111, 112]. Additionally, for a constant amount of development effort, N-version programming is actually less reliable than focusing on a single version over a wide range of parameter values [18].

Another reliability mechanism is the recovery block concept [113]. In this approach, several alternative methods are developed. We first run the fully featured one and check if it is correct. If it is, we use it. Otherwise, we try the simpler ones. The essential difference between recovery blocks and the Simplex architecture is that the former is a backward recovery method while the latter is a forward recovery method.

A common engineering practice to increase system reliability in spite of unreliable hardware is triple modular redundancy (TMR) [96]. In this scheme, three versions of identical hardware
The switching set of points output by our algorithm is shown projected onto fixed \( v \) and \( \theta \) dimensions. Here, a single segment between two waypoints (crosses) is analyzed. The light gray states indicate the unsafe regions, whereas the dark gray states are states where the safety controller must be immediately used in order to maintain safety. The top figure is for \( v = 1000 \text{mm/s} \) and bottom figure is for \( v = -1500 \text{mm/s} \). In both figures, \( \theta \) is fixed at 0.

running an identical program are run with the same input. The output is then voted upon, such that if any one of the outputs is incorrect (due to a hardware failure or random environmental interference [114]), the overall system continues to function correctly. This technique, unlike the System-Level Simplex Architecture, is powerless against errors in the logic of the program, since all three modules will produce the identically incorrect output. However, it is effective against hardware failures and can be used in conjunction with the System-Level Simplex Architecture. The resultant architecture prevents logical errors, transient faults, and hardware failure problems.

To use this combined scheme, we would have three modules each with their own hardware and software portions running the System-Level Simplex Architecture with a reliable voter at the end to accumulate the results. Variations of this also are possible, for example by replicating only the safety-critical hardware subsystem and using a single microprocessor-based complex controller.

System-Level Simplex is most closely related to the original Application-Level Simplex Architecture [18, 115, 116]. In this design, two controllers are used in software to provide reliability in spite of logical errors in the complex version. The System-Level Simplex is a novel architecture over the Application-Level Simplex in several ways. The System-Level Simplex Architecture eliminates a large body of common unverified dependencies between the safety and complex controllers, in-
cluding the operating system, middleware, and microprocessor. Additionally, moving logic outside of software allows us to handle additional failure modes previously unavailable to software running within the Application-Level Simplex design, such as power and timing faults of the microprocessor-based system. The System-Level Simplex also removes computation overhead from the processor, which no longer has to run the simple controller and decision module (which could affect real-time schedulability). One drawback of the System-Level version of Simplex is that additional resources are required to run the decision module and complex controller, such as the FPGA.

Simplex (both versions) should not be regarded as a one-size-fits-all robustness approach. Even if a verified decision module is produced, there is likely pessimism in the switching and the simplicity of the safety controller can unnecessarily reduce system performance if it is used too often. This is one trade-off a Simplex system makes in order to guarantee robustness and controllability.

Quantization-Based Discrete Abstractions

Finding finite state abstractions for hybrid automata is a problem of fundamental importance in design and verification of cyber-physical systems. This is because the availability of a bisimilar, finite-state abstraction \( B \) for a hybrid system \( A \), makes it possible to design and verify \( A \) by solving similar problems for \( B \). This is desirable because the latter problem can be attacked with powerful model checking tools, such as Maude [108] or SMV [117]. Techniques for abstracting hybrid automata have been sought after since the inception of the field, and it was quickly found that finite-state, bisimilar abstractions may not exist for general hybrid systems [103, 118, 119]. Nevertheless, restricted classes of hybrid automata have been identified for which such abstractions exist and can be algorithmically constructed [120, 121]. Unfortunately, the classes of hybrid automata that arise from the types of vehicular systems we are interested, generally do not fall within these restricted classes.

Apart from bisimulation-based, finite-state abstractions, ordinary simulation-based abstractions have been used [122, 123] for deductive verification of hybrid systems. Most of these techniques involve significant manual work in finding the simulation relation that relates the concrete hybrid automaton \( A \) with the (simpler) abstraction \( B \).

Several related notions of abstractions based on approximate bisimulation relations have been proposed [124, 125, 126]. One work particularly relevant to our paper [126] shares a similar goal—that of constructing finite abstractions for designing (or synthesizing) controllers with certain properties. One difference, however, is that we work with traditional simulation relations that give a partition of the state space, instead of approximate simulations which give a covering of the con-
tinuous state space. The general problem of synthesizing switching controllers that are correct by construction has been looked at earlier, mostly for linear and piece-wise affine systems [127, 128]. To the best of our knowledge, however, the existing work does not address differential inclusions and periodically-switched controllers.

Earlier work has also used the notion of computing back reachability to construct correct-by-design switched modules in Simplex-like systems [129]. However, the earlier proposed approaches assumed control could instantly switch to the safety controller, rather than periodic sampling, and therefore did not contain $\Delta$-time bounded backreachability using the complex controller dynamics.

Two Step Approach for Reachability

There are several algorithms and tools for computing reachable states and approximate reachable states for hybrid systems [130, 131, 132, 133, 134, 135].

Work related to the PESSOA [136] tool for synthesizing embedded controllers is similar in spirit to our work. PESSOA generates a finite state abstraction of a given system and uses these abstractions for synthesizing controllers that are guaranteed certain restricted class of LTL properties such as $\Box \varphi$, $\Diamond \varphi$, $\Diamond \Box \varphi$ and $\Diamond \Box \varphi \land \Box \varphi'$. The controller synthesis uses earlier techniques [137, 138, 139, 140]. The finite state abstractions used in PESSOA are approximate simulations [141, 124] of the original continuous system. In contrast, the finite state abstractions we use are simulations of the original system in the classical sense. Furthermore, our algorithm (and tool) can handle plants which are described with hybrid automata using a class of nonlinear differential equations. To the best of our knowledge, PESSOA currently does not support such models.

Checkmate [142] is a tool for verifying control systems modeled as a hybrid automaton. Checkmate computes the reach set for a linear and non-linear systems by using the flow pipe approximation technique [143] to approximate the reachable sets by a sequence of convex polyhedra. Checkmate restricts the class of hybrid systems that can be verified to polyhedral-invariant hybrid systems. Our approach, however, can verify hybrid systems without this restriction. Additionally, correctness in Checkmate requires solving a global optimization problem, for which there is no general sound solution.

118
4.4 Conclusions

We have presented the System-Level Simplex Architecture which uses hardware/software co-design to produce fault-tolerant systems. By leveraging on a simple safety controller and a decision module implemented in hardware, several types of previously unhandlable errors can be safely managed. These include failures in the complex software controller code, operating system, and microprocessor, as well as real-time temporal faults.

An end-to-end design process has been created for the architecture, which leverages on an initial AADL model to provide both an architectural and behavioral description. The output of the process is a checkable System-Level Simplex architectural model, and the corresponding VHDL hardware code.

We demonstrated the feasibility and robustness of the System-Level Simplex Architecture through two case studies involving a pacemaker and a classic inverted pendulum. The architecture is also currently being evaluated to improve the safety of autonomous tractor control, in collaboration with John Deere.
In this work, we have attempted to address two critical challenges in cyber-physical system design: safety and affordability. We first considered the global coordination of the CPS task, and provided a necessary and sufficient condition for predicates to be invariants of a system. This was done assuming a realistic model of the communication, where packets can be delayed or dropped. This model is applicable to both wireless communication, as well as communication performed over the Internet. Next, we outlined challenges to affordability in CPS systems. Specifically, using affordable commercial-off-the-shelf (COTS) components usually results in poor real-time performance which is necessary for tasks which interact with the physical world. We addressed memory-access timing issues with COTS components caused by I/O peripherals, and caused by multiple CPU cores concurrently competing for memory access. Finally, we addressed safety for the individual nodes of a distributed CPS by using the Simplex architecture. Here, we provided algorithmic ways, based on reachability computations of hybrid automata, to generate the logic of the Simplex decision module, which is responsible for switching to the backup safety controller before safety violations can occur. The research presented here is ongoing, and we plan to expand on each of the areas we have presented.

In terms of the global CPS safety, we have shown which conditions are necessary and sufficient for safe system design. We plan to use this approach on a more complicated case study, where the safety actions do not consist of disjoint paths, but rather involve nontrivial assumptions about the other agents. One example of such a system is an intelligent intersection [25], where cars may continue to proceed through the intersection if communication is delayed or lost. Another direction here would be to find algorithms which assume the same communication model, and check if computing the proposed infinite-time reachability condition if feasible for such systems.

As presented in this work, we have started to address the challenge of applying the PRedictable Execution Model (PREM) to multicore systems. Here, the challenge is not necessary rearranging memory access as we have already done, but rather scheduling access to memory in such a way
that system utilization remains high. We plan to mathematically analyze the M-LAX scheduling algorithm which we have evaluated in simulation, and compare the formal guarantee to TDMA-based Memory Centric Scheduling. Other approaches have also being recently developed for the problem of memory interference in real-time systems, such as MemGuard which throttles memory access from each core to prevent delays due to main memory saturation [144]. A comparison of this approach with multi-core PREM is planned.

We have also started to generalize the presented algorithms for reachability for hybrid automata into general non-linear systems which do not have restrictions on the dependency graph imposed by the system’s differential equations, using a technique similar to mixed face lifting [145]. This approach has successfully computed unbounded reachability for low-dimension systems with non-linear coupled dynamics. We would also like to improve the scalability of the approach into a higher number of dimensions, especially for well-behaved systems where the reach set is convergent or is cyclic. Another direction is to have an explicit unsafe set, and use counter-example guided abstraction refinement (CEGAR) to focus on the areas of the computation where a safety violation may occur, and accept larger overapproximation errors in other areas of the state space where they do not matter.
Bibliography


