ABSTRACT

Title of dissertation: RUNTIME ADAPTATION IN EMBEDDED COMPUTING SYSTEMS USING MARKOV DECISION PROCESSES

Adrian E. Sapio, Doctor of Philosophy, 2019

Dissertation directed by: Professor Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park

During the design and implementation of embedded computing systems (ECSs), engineers must make assumptions on how the system will be used after being built and deployed. Traditionally, these important decisions were made at design time for a fleet of ECSs prior to deployment. In contrast to this approach, this research explores and develops techniques to enable adaptation of ECSs at runtime to the environments and applications in which they operate. Adaptation is enabled such that the usage assumptions and performance optimization decisions can be made autonomously at runtime in the deployed system.

This thesis utilizes Markov Decision Processes (MDPs), a powerful and well established mathematical framework used for decision making under uncertainty, to control computing systems at runtime. The resulting control is performed in ways that are more dynamic, robust and adaptable than alternatives in many scenarios.

The techniques developed in this thesis are first applied to a reconfigurable embedded digital signal processing system. In this effort, several challenges are encountered and resolved using novel approaches. Through extensive simulations and a prototype implementation, the robustness of the adaptation is demonstrated in comparison with the prior state-of-the-art.

The thesis continues by developing an efficient algorithm for conversion of MDP models to actionable control policies — a required step known as solving the MDP. The solver algorithm is developed in the context of ECSs that contain general purpose embedded GPUs (graphics processing units). The novel solver algorithm, Sparse Parallel Value Iteration (SPVI), makes use of the parallel processing capabilities provided by such GPUs, and also exploits the sparsity that typically exists in MDPs when used to model and control ECSs.

To extend the applicability of the runtime adaptation techniques to smaller and more strictly resource constrained ECSs, another solver — Sparse Value Iteration (SVI) is developed for use on microcontrollers. The method is explored in a detailed case study involving a cellular (LTE-M) connected sensor that adapts to varying communications profiles. The case study reveals that the proposed adaptation framework outperforms a competing approach based on Reinforcement Learning (RL) in terms of robustness and adaptation, while consuming comparable resource requirements.

Finally, the thesis concludes by analyzing the various logistical challenges that exist when deploying MDPs on ECSs. In response to these challenges, the thesis contributes an open source software package to the engineering community. The package contains libraries of MDP solvers, parsers, datasets and reference solutions, which provide a comprehensive infrastructure for exploring the trade-offs among existing embedded MDP techniques, and experimenting with novel approaches.

RUNTIME ADAPTATION IN EMBEDDED COMPUTING SYSTEMS USING MARKOV DECISION PROCESSES

by

Adrian E. Sapio

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Table of Contents

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible. First, I would like to thank my advisor Professor Shuvra Bhattacharyya, for giving me the opportunity to work on challenging and interesting projects over the past years. I would also like to thank Professor Marilyn Wolf, whose insight and guidance on various matters proved very helpful along the way.

The deepest thanks of all go to my wife Mayrola, for without her support and sacrifice over a countless number of nights and weekends this work would never have been completed. Finally I would like to thank my parents, for sending me off in the right direction and providing unconditional support.

This research was supported in part by the Computer and Network Systems Program of the U.S. National Science Foundation under Grant No. CNS1514425 and CNS151304.

Chapter 1

Introduction

This thesis presents techniques developed to achieve runtime adaptation in resource constrained embedded computing systems (ECSs). Adaptation is defined herein as strategic decision-making in the control of one or more of the following: algorithm selection (among a set of available algorithms), parameter selection from a set, and partial or complete reconfiguration of programmable digital logic from a set of available logic configurations.

This work seeks to leverage the computational and reconfiguration capabilities of modern embedded computing platforms to develop systems that can better adapt to the environment in which they are operating. Adapting to the environment using an effective *system-level reconfiguration framework* (*SLRF*) can help these systems operate more effectively — e.g., with improved trade-offs among achievable data rate, latency, and energy efficiency.

During the design of embedded computing systems, engineers must make assumptions on how the system will be used after being built and deployed. These assumptions are typically used to balance multiple conflicting performance objectives, and they have a significant effect on how well the system ultimately performs. This research seeks to explore and develop techniques to help advanced systems better adapt at runtime to the environments and applications in which they operate. The goal is to embed novel algorithm implementations that can be deployed alongside the application functionality, with the purpose of enabling the system to continually reconsider and update its own operational trade-offs autonomously.

The framework includes multiobjective optimization at its core, embracing the multifaceted nature of embedded computing design, where making strategic trade-offs among conflicting goals is critical. Additionally the framework allows for accumulated knowledge and beliefs to be represented probabilistically, which is a crucial requirement since the real-world environments in which embedded computing systems operate are typically not deterministic.

The alternative to runtime adaptation is systems in which the runtime behavior is fully specified at design time, prior to deployment. The disadvantage to this approach is that it forces engineers to make assumptions about the environments that their systems will be deployed in, which is often not feasible, or not practical for widely deployed systems that encounter a wide range of environments — e.g., a cellular connected device that might have one unit deployed in a location with very strong cellular coverage and another with weak cellular coverage.

In this scenario, if the system performance could be optimized by a cellular communications modem configuration, an engineer might be forced to select one configuration for both installations that would be either better suited for one over the other, or a compromise between both that would be sub-optimal in either case. As thousands or even millions of embedded computing systems are deployed in ubiquitous ways throughout all areas of society, a manual optimization of each individual unit of this form becomes infeasible. If on the other hand, a runtime adaptation framework is used, the systems can be programmed to self-optimize towards high-level goals such as maximizing battery life and maintaining a reliable communication link. In this way, they can autonomously make effective trade-off decisions for each individual installation, after being deployed and even in response to changing conditions over time. Environmental properties such as the quality of cellular coverage could have a short term, time-dependent variation such as congestion during peak hours of the day. Environmental properties could also have a long term, time-dependent variation such as improvement due to new cellular towers being installed, or congestion as a community grows and more users populate the coverage area.

The design of algorithms to control resource constrained computing systems effectively at runtime has been a topic of active research for at least 20 years. A good survey for early work in this area can be found in [1]. This survey reviews a wide range of techniques, including fixed threshold-based approaches, dynamic approaches including stochastic controllers using dynamic programming, as well as some guidance on how MDPs can be used in this context.

Since the time period when that survey was written, a variety of approaches to this research problem have flourished over the years. Some researchers have sought to formulate the design challenge as a constrained optimization problem [2]. Other researchers focused on modeling the dynamics of the system's energy consumption, and simplifying the control decisions to be simple threshold-based comparisons with respect to the energy budget (e.g., see [3, 4]). Another popular approach has been to model the system as linear in the context of feedback control systems and then use Model Predictive Control (MPC) theory to modulate a processing duty cycle (e.g., [5, 6]).

All of these approaches were shown to be successful in their respective case studies, but share some common limitations when considered for use in other cases. For example, several of these approaches assume deterministic behavior from the system being controlled. These approaches model the behavior of the system in response to some actuation, and assume the system will always behave the same way. However, many ECSs have some stochastic behavior, either due to complex unmodeled dynamics or due to being affected by an external factors that are difficult to predict. A second common limitation is the assumption that the system being controlled can be modeled as a linear system. Computing systems often do not behave in linear ways, and attempts at formulating linear approximations to non-linear behavior is limited to only the simplest non-linearities, which significantly constrains the overall applicability and generality of this approach. A third common limitation is that the dynamics of the system being controlled often need to be well understood at design time. For many computing systems whose behaviors depend heavily on external factors, this can be an unrealistic assumption.

As efforts in this area progressed, the paradigms shifted from classical control systems theory to various forms of adaptive algorithms, and then to more generalized approaches that researchers have termed as self-configuration, self-optimization and most recently, self-awareness [7]. A wide ranging survey of these works and organization of them into these various self-X categories can be found in [8]. In that work, researchers define self-awareness as "attributes in a system that enable it to monitor its own state and behavior, as well as the external environment, so as to adapt intelligently". Another definition can be found in [9]: "self-aware computing describes a novel paradigm for systems and applications that proactively maintain knowledge about their internal state and its environment and then use this knowledge to reason about behaviours".

Among the most promising directions for creating these self-awareness attributes in ECSs is through the use of Markov Decision Processes (MDPs), a powerful and well established mathematical framework used for decision making under uncertainty. The SLRF proposed in this thesis makes use of MDPs, as they have shown success in this area because they are inherently capable of modeling stochastic behavior and non-linear responses, and they are also well equipped to deal with incomplete models and uncertainty. This allows decisions to be encapsulated into a policy-based framework, where the decision options and the criteria for making those decisions are defined and the adaptation problem is shifted to an optimal policy design problem.

Following this approach, the design effort can be decomposed into at least the following sub-problems, which must be addressed in the design of the proposed form of embedded decision agent.

- Modeling the available system capabilities in terms of how they help to accomplish the high-level goals.
- Determining trade-offs associated with these capabilities.
- Modeling the effects that the exogenous environment has on these goals.
- Predicting the future behavior of the environment based on past statistics, and anticipating the effectiveness of alternative available functionalities in the future environment.

In this thesis, a methodology is presented that addresses these sub-problems, analyzes the effectiveness of the resulting adaptive systems, compares the performance to competing approaches, and explores numerous issues that arise in the process.

In Chapter 2, the methodology is applied to the control of digital filtering and decimation operations commonly found in the physical layer Digital Signal Processing (DSP) of wireless communication systems. In this work, several challenges were encountered and resolved using novel approaches. These include the use of transition states to accurately model a wide range of the processing system dynamics, a scalarization approach to employ the MDP within a multiobjective optimization framework, and the factorization of system states into internal and external groupings for efficient representation and embedded computation. Additionally, the concept of factorization is applied to reduce the storage size of the MDP model and execution time of the policy generation algorithms.

The limitation of the work in Chapter 2 is that the policy that is analyzed is an immutable policy that is generated once, offline. There is no runtime adaptation in this approach, however the work serves to establish the feasibility of the framework as a viable tool for runtime adaptation of the stated task. Continuing in this spirit, the work presented in Chapter 3 begins addressing the challenges of deploying time-varying control policies under the MDP-based framework. The work shows that sparsity — a characteristic of MDPs that arises when they are used in this context — is an important concept to exploit for efficient and practical policy generation within embedded computing systems at runtime. Then, the processing parallelism capabilities of modern embedded graphics processing units (GPUs) — such as those used in smartphones, robotics and automobiles — are leveraged to generate control policies in faster and more efficient ways than the prior state-of-the-art.

Chapter 4 extends the applicability of the runtime adaptation methodology to a wider class of embedded computing systems. This extension is achieved by developing a variation on the approach that is suitable for smaller microcontroller-based systems (without GPUs), and then presenting a detailed design and implementation of an adaptive controller in such a system. The controller is compared to Reinforcement Learning (RL) based techniques on a range of performance criteria. The comparison of the proposed method with RL-based techniques is generalized, exploring when one approach is better suited over the other, and also exploring the system aspects that affect this choice.

Following these developments, Chapter 5 contains detailed surveys and a reflection on the state of affairs encountered in the endeavors of Chapters 2-4. Limitations and hindrances that were found are identified, such as the lack of available benchmarking data, and widespread incompatibility between MDP file formats. In an attempt to remedy some of these issues for future research efforts, a common embedded MDP development platform is proposed by authoring an open-source software package and pairing it with a family of popular off-the-shelf development hardware boards. The software package has recently been released to the academic community, and feedback has been solicited from communities of interest.

Finally, Chapter 6 contains conclusions from this research and also discussion on areas for future work.

Chapter 2

Reconfigurable Digital Channelizer Design Using Factored Markov Decision Processes

2.1 Introduction

Digital channelizers are critical subsystems in wireless communication systems that are employed when a multiplexed signal contains information in different frequency subbands, and the application requires separating one input signal (containing multiple subbands) into one or more output signals (each containing a subset of the input subbands) [10]. This function is commonly required in cognitive radio systems [11, 12]. In this chapter, the reconfiguration capabilities of modern embedded platforms are leveraged to develop digital channelizers that can better adapt to the environment in which they are operating. The material in this chapter was published in preliminary form in [13, 14].

Adapting to the environment using an effective *system-level reconfiguration framework* (*SLRF*) can help these systems operate more effectively — e.g., with improved trade-offs among achievable data rate, latency, and energy efficiency. For this purpose, MDPs are applied in novel ways to make dynamic decisions on maintaining or adapting signal processing configurations during channelizer operation. An MDP-based SLRF is proposed to develop dynamic reconfiguration policies for use in stochastic environments in which adaptation of hardware/software configurations for digital channelizer processing is strategic.

While the SLRF techniques are developed in this chapter with a specialized focus on digital channelizer implementation, the results suggest that the underlying MDP techniques are applicable across many other types of embedded signal processing systems (ESIPs). Exploring the generalization of the SLRF for broader classes of ESIPs is therefore a useful direction for future work.

The MDP-based approach for digital channelizer design optimization results in increased robustness when used to periodically re-optimize the system policy specifically for the external environment it is being used in. This periodic re-optimization can be done completely autonomously by an embedded signal processor, without any need for human-in-the-loop intervention. The information that these design optimization methods require is completely observable by the system at runtime.

The results in this chapter show that MDPs are useful tools for controlling resources in computing systems. Additionally, two innovations are introduced that significantly enhance the effectiveness of MDPs for channelizer design optimization. First, a mechanism is added to address hardware/software codesign scenarios that involve multidimensional design objectives and constraints, which are commonly encountered in transceiver system design. This is done through a multidimensional framework for the definition of the MDP rewards function.

Second, a concept called *transition states* is introduced to represent intermediate states (between distinct channelizer configurations) in the target system. The transition states are applied in scenarios where commanding a state change can result in one or more time steps (frames) where the system is in a non-productive transition mode. Since being in transition from one state to another can result in missing real-time deadlines for processing requests, the control policy must choose carefully when to command a transition, and only seek to do so when the end result will be a net positive for the system in the long run, in spite of any short-term negative effects due to the transition frames. Such incorporation of transition states within the SLRF extends its utility to a broader class of applications, including channelizers, where transitions between productive states must be taken into account for accurate assessment and optimization of dynamic reconfiguration control.

The work in this chapter continues by applying a methodology developed in [15] to transform an MDP into a *factored* MDP. This concept addresses a problem that frequently occurs with MDPs — the number of possible states of the model can be extremely large. As detailed in [16], a major motivation behind factored representations is that some parts of this large state space generally do not depend on each other and that this independence can be exploited to derive a more compact representation of the global state. In this thesis, factorization serves to reduce the storage size of the MDP model and execution time of the policy generation algorithms. Such advancements are critical enablers for the direction of this work — deploying the modeling framework and policy generation algorithms on embedded systems.

When the framework and algorithms are integrated with the application on an embedded platform, they can be used to perform periodic re-optimization of the reconfiguration policies in addition to applying the policies to manage system configurations. To be practical in resource constrained and power constrained embedded environments, the deployed implementations of the modeling framework and policy generation algorithms must be carefully optimized so that they consume minimal amounts of storage and impose minimal computational burden. The application of factored MDP techniques in this chapter is an important step towards these objectives.

Next, the findings of an expanded performance analysis of the proposed methodology are detailed. Specifically, a suite of competing control policies are described and compared objectively with the MDP based techniques. The results show that the MDP based techniques outperform the alternative schemes in nearly all cases.

Finally, a trade-off analysis of the costs and benefits of including transition states in the framework is presented. This exploration details and quantifies the design time modeling costs of transition states in both storage size and execution time. These costs are then contrasted with the benefits in the form of the run-time performance when transition states are included versus when they are not.

The remainder of the chapter is organized as follows. A cursory review of the history of channelizers and MDPs, and their development is presented in Section 2.2. In Section 2.3, the signal processing application and the algorithms involved are detailed. In Section 2.4, the MDP-based approach is introduced, along with an illustration of how it is applied to the signal processing application. That is followed by Section 2.5 with a summary of the simulations performed and the resulting data and observations that were made. The chapter concludes in Section 2.6 with a discussion of future work on the use of MDPs in channelizers.

2.2 Background and Related Work

A digital channelizer can be generalized as having the inputs and outputs shown in Figure 2.1. Without loss of generality, the inputs and outputs are represented as frame-based vector quantities, with time decomposed into fixed-width slots referred to as *frames*. The frame arrival rate is constant and the stream of incoming frames is never ending. A channelizer is often a subsystem of a larger signal processing system. For each frame of data, the channelizer is commanded by higher-level elements of the larger signal processing system on a per-frame basis. These higher level elements determine which subchannels need to be produced and which do not.

An example of such a channelizer framework can be found in the cognitive radio of [17]. In that application, a channelizer is used to isolate sub-bands within some wireless spectrum dynamically. This dynamic behavior involves consuming a wideband signal, and applying digital filters and rate-changing operations to produce an output that contains some subset of the input signal frequencies.

In Figure 2.1, for each frame n of data, $x^{(n)}$ is a length N complex vector of the wideband input signal. This data is presented to the channelizer alongside $CR^{(n)}$, a length N_C binary vector that provides the channelization request for that frame. The channelizer outputs N_C parallel output data vectors,

$$y_{\alpha}{}^{(n)}, \alpha = \{1, 2, \ldots, N_C\}. \tag{2.1}$$

Each of these vectors contains a channelized subset of the input.

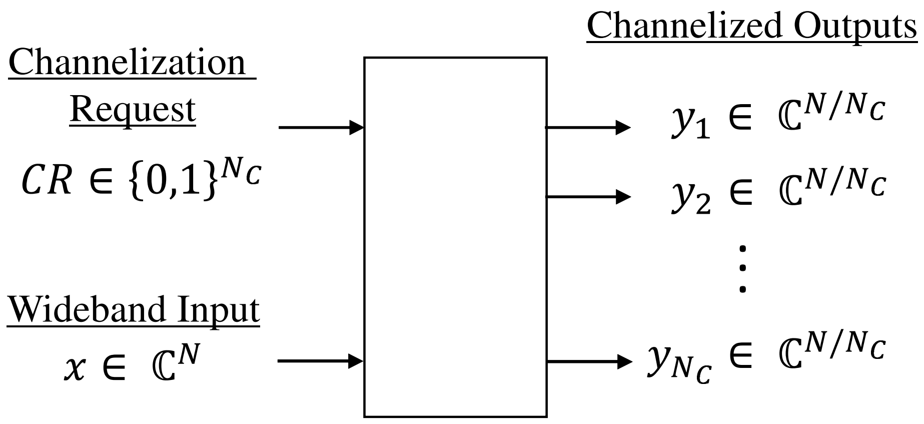

Figure 2.1: Channelizer inputs and outputs.

Good surveys of popular digital channelizer architectures to date are found in [10, 18, 19]. The most common architectures are based on the Cosine Modulated Filter Bank (CMFB), Discrete Fourier Transform Filter Bank (DFTFB) and Per-Channel Filter Bank (PCFB). Aside from these well-established architectures, several other interesting designs for application-specific channelizers can be found in [20, 21, 22, 23].

As illustrated in [11, 24], the channelizer is often one of the most computationally intensive and power consuming blocks of cognitive radio transceivers, mainly due to its need to run at the highest data rates. For this reason, several researchers have sought to create channelizer designs where the key parameters that control the processing (e.g., filter coefficients, data rates, and subchannel masks) are configurable at run-time [25, 26, 27]. This class of DSP systems is referred to as "reconfigurable channelizers", and the active body of DSP research is evidence of the importance of optimizing channelizer processing for exactly what is required, and nothing more. The goal is generally to improve efficiency by increasing processing productivity, while simultaneously decreasing energy consumption.

The body of prior work referenced above provides a number of efficient channelizer designs that can be flexibly configured for different trade-offs. However, this body of work does not address how or when the configurable parameters are changed, nor provide policies for changing them at run-time. In this chapter, MDP-based methods are developed to bridge this gap.

Other researchers have sought to use MDPs with similar goals. Wei et al. have demonstrated the effective use of an MDP to control the processing rate of a network router [28]. This work created a Markov model of only the external environment, not the system under control. In contrast, the SLRF proposed in this chapter incorporates Markov models of both the controlled system and the external environment, which provides a more comprehensive foundation for dynamic adaptation.

Hsieh et al. [3] devise a scheduling policy that selects among alternative implementations of common functions, such as FFTs. The alternative options accomplish functionally the same operation, but with different execution times, power demands, and hardware requirements. As in the SLRF proposed here, Hsieh's approach uses an algorithm to make reconfiguration decisions based on what requests are placed on the system at runtime. However, in Hsieh's approach, these requests are converted to a time series signal, smoothed using a moving average filter, and then compared to thresholds in order to derive reconfiguration decisions. The designer must commit to a smoothing factor on the incoming requests, and effectively assume a priori some of the resulting dynamics of the system.

Compared to Hsieh's methods, the SLRF proposed here takes a very different approach by transforming both the system and operating environment into stochastic models, which can then be reasoned upon within the framework of MDPs. In contrast to the approach of Hsieh, there are no a priori trade-offs on the smoothing of incoming requests. Furthermore, instead of condensing the observable data into one-dimensional signals, larger conditional probability tables are maintained. Thus, the algorithms in the SLRF can incorporate more knowledge into the decision framework. By incorporating historical transition probabilities, the MDP is able to infer in real-time whether a new request is likely to be the start of an event that should be acted upon, or is more likely a spurious request that is better ignored. This inference can be performed immediately and without the delay associated with the step response through a smoothing filter.

2.3 Reconfigurable Channelizer Design

In this section, a reconfigurable digital channelizer design is presented, that forms the foundation for an MDP-based, adaptive channelization system. The system is detailed in Section 2.4, and demonstrated experimentally in Section 2.5.

The channelizer system is implemented on the Silicon Labs EFM32GG, a small and low power ARM Cortex M3-based microcontroller. The processor is running on the EFM32 STK3700 development kit, which houses the CPU as well as sophisticated energy monitoring circuitry. For this hardware, a channelizer width of $N_C = 8$ subchannels is used in an illustrative experiment.

This particular channelizer system is developed with applicability to wireless sensor networks, which impose challenging constraints on energy consumption and resource utilization. However, with its foundation in MDP techniques, the design methodology is not specific to any particular domain of channelization applications. For example, the methodology can be adapted to large scale, high performance channelization scenarios that involve dozens or hundreds of subchannels that require the use of FPGAs or GPUs to run in real-time. Developing such adaptations for these additional classes of processing platforms is an interesting area for future investigation.

To examine the ability of the system to adapt to its environment, two separate use cases are considered, which are referred to as A and B. Additionally, multiple scenarios are created within those use cases, by varying parameters of the application that are understood to be time-varying. Two separate channelizers are designed, one ideally suited for each use case, as detailed in Section 2.3.1 and Section 2.3.2. A reconfiguration policy is then derived using the SLRF with the decision-making authority to select which channelizer algorithm to use at any given time. Additionally, the algorithms contain configuration parameters, and the SLRF is given control of these parameters. This results in a unified controller for reconfiguration, dynamic power management, and runtime parameter optimization.

2.3.1 Polyphase DFT Filter Bank

Use Case A is the application in [17]. In that system, the requests are modeled as i.i.d. (independent and identically distributed) Bernoulli across both the time and subchannel dimensions. These statistics for the requests mean that there is no opportunity to anticipate the request vector. For such an environment, a sensible option is a filter bank that outputs all subchannels at all times, in the most efficient way possible. For this, a Polyphase implementation of the canonical Discrete Fourier Transform Filter Bank (DFTFB) described in [10] is used.

To implement this DSP block, a low pass filter is designed to be used as the “prototype” filter in the filter bank. The filter has a passband width of one eighth of the full spectrum, since there are eight equally spaced channels. The filter coefficients are chosen using the Equiripple FIR design method detailed in [29]. The prototype filter is then shifted in frequency, decomposed into its polyphase components $E_m(z)$, and implemented into the DFTFB, as described in [10]. A block diagram of the derived DFTFB is shown in Figure 2.2. The resulting magnitude

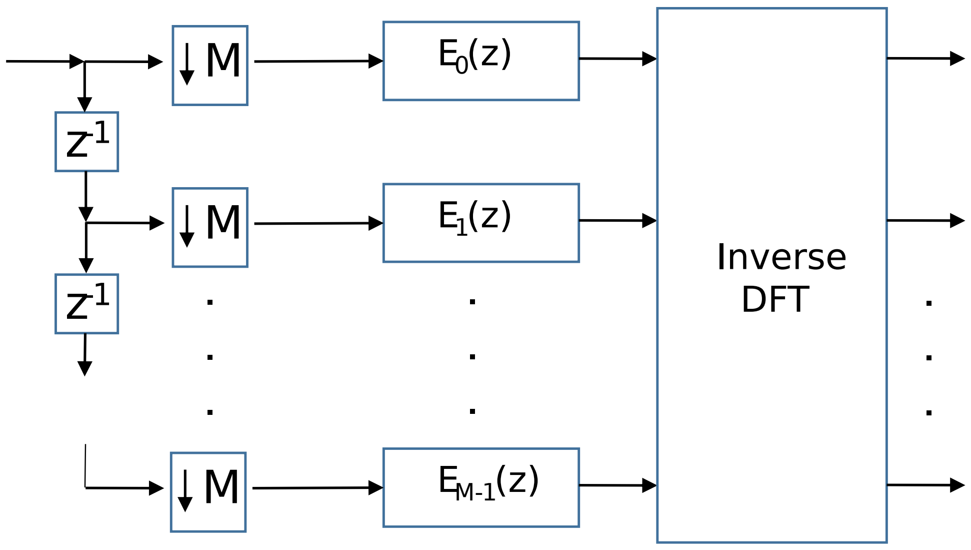

Figure 2.2: DFTFB block diagram, $M = N_C$.

response for each of the 8 outputs is shown in Figure 2.3.

As can be seen from the magnitude responses of the 8 channelized outputs, this filter bank can simultaneously channelize all of the subchannels, and thus, no tunable parameters are required for this algorithm. In order to optimize for bursts of communication activity as well as idle time, the controller has the ability to put the DFTFB in and out of a *sleep mode*. The DFTFB remains resident in the current configuration, and can be gated on and off very quickly. The gating off of the DFTFB corresponds to its sleep mode.

2.3.2 Tunable Polyphase Decimation Filter

Use Case B is the Sequential Sensing application in [30], where a channelizer with the same inputs and outputs as Use Case A is required. However, the request statistics imposed on this channelizer are quite different from those in Use Case A.

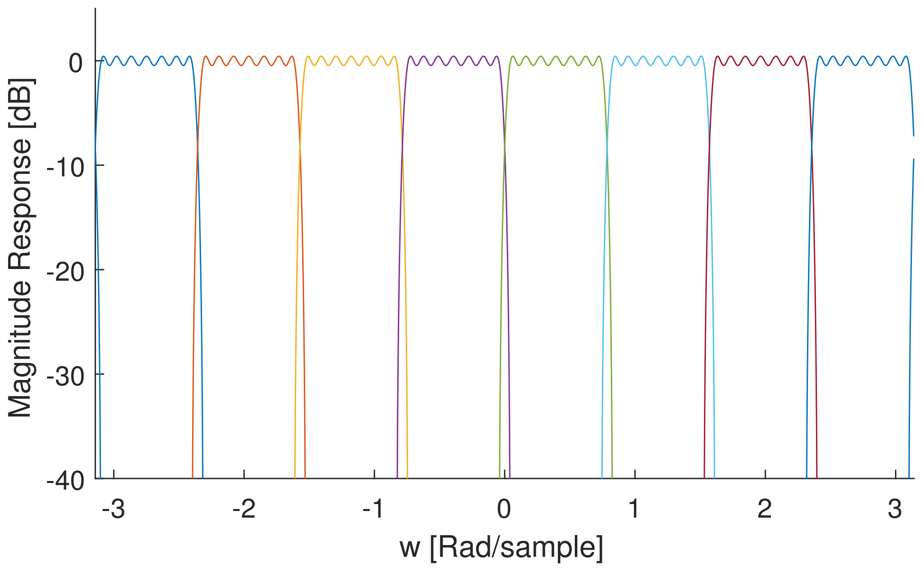

Figure 2.3: DFTFB magnitude responses.

In Use Case B, the channelizer is requested to produce only one output subchannel at a time. One or more frames (usually multiple frames) elapse between requests for different subchannels.

Since only one channel is requested at any given time, only a tunable decimation of the input data is needed — i.e., to filter out the unwanted subchannels. For this, a polyphase implementation of an 8-to-1 decimation (DCM) filter and mixer as described in [31] are employed, shown in Figure 2.4.

The operation shown suppresses all but one subchannel out of the incoming signal, and then uses a complex mixer to shift the extracted channel down to be centered at DC. Once centered at DC, a simple decimation of samples gives the resulting output stream. The same filter coefficients used for the prototype low pass filer of the DFTFB can be used in the DCM. Such a DCM design produces the same frequency response per subchannel. Prior to implementation, the polyphase

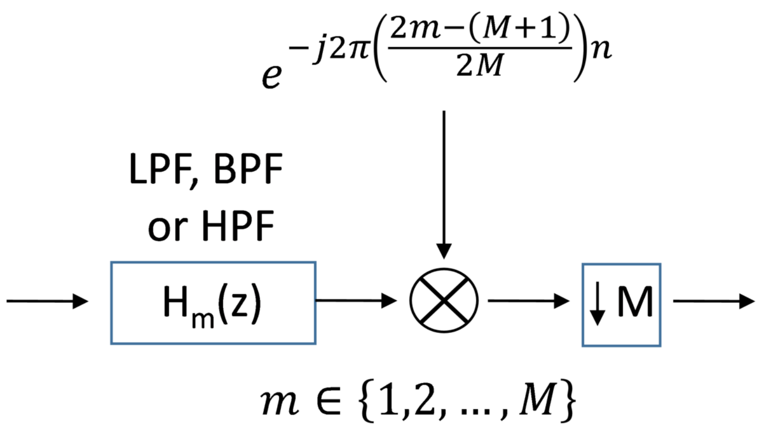

Figure 2.4: DCM block diagram, $M = N_C$.

technique detailed in [31] is utilized to reduce the runtime processing requirements further without changing the resulting filtering operation. The resulting subsystem is referred to as a *polyphase decimation filter*.

Unlike the DFTFB, this configuration does have tunable parameters: the filter coefficients and mixing frequency. Using 8 parameter sets, this algorithm can be modified to select any of the 8 subchannels, effectively being an efficient low-pass, band-pass or high-pass decimation filter. Both the filter coefficients and the amount of frequency shifting are tunable, as shown in the block diagram (Figure 2.4). The signal is first passed through a digital filter $H_m(z)$, whose coefficients are specific to each channel m. Then, the filter output is shifted in frequency by multiplying it with a complex sinusoidal signal, whose frequency is also specific to each channel m. The formula to generate the sinusoidal frequency is the exponential shown in the block diagram. This configuration is also designed to be kept in a sleep mode during periods of idle user activity.

An important implication of using digital filtering algorithms based on Finite Impulse Response (FIR) filters and not Infinite Impulse Response (IIR) filters in the DFTFB and DCM processing configurations is that the resulting signal processing algorithms are numerically stable, unconditionally. IIR-based designs by necessity require careful positioning of the transfer function's poles, as well as attention to how those pole positions are affected by the numerical implementation of the filter in terms of quantization effects, fixed point field widths and the dynamic range of intermediate calculations. A thorough analysis of these concepts can be found in [29].

2.3.3 Summary of Processing States and Their Properties

The MDP framework requires an enumeration of the states that the processing system can be in at any time. The experimental embedded system has 13 states, which fall in the categories listed in Table 2.1.

The first row of the table covers the states when the system is in a sleep mode, with either the DCM or DFTFB ready to run. The distinction is made between these as two separate states to allow the model to capture any difference in time that it may take to re-enable the resident and already initialized algorithm out of sleep mode compared to switching to the other algorithm. Further discussion on these delays will be presented in Section 2.4.3.

The last two states, whose labels are prefixed with "Trans.", are states of being *in transition* to the DFTFB or DCM, respectively. The time required by the processing system to transition between states is an important detail in this framework. The incorporation of transition states into the MDP is a novel contribution in this work that is intended to take such transition times into account (detailed in Section 2.4.3). This concept of transition states allows an SLRF to compute decision paths involving transitions that can take multiple time frames to complete.

The third column of the table shows the number of channels provided by the system while in each state. Note that while in transition, the system is consuming power but not producing any channelized data.

The fourth column of the table shows the CPU power consumed by the system in each state. These measurements were performed at design time by putting the processor into test modes created for this purpose. Each test mode loaded a single configuration and executed it at the experimental application's frame rate. With the processor operating in such a test mode, the Silicon Labs EFM32GG development tools allowed the power consumption of the associated state at the associated frame rate to be measured.

It is clear from Table 2.1 that the DFTFB is the most productive configuration (producing all 8 subchannels), while being the most power hungry in its ON state. It is also clear from the table that the DCM algorithm represents a less productive configuration (producing only 1 subchannel) compared to the DFTFB, but with the benefit of reduced power consumption. If only one channel is requested for an extended period of time, then a rational controller should select the DCM configuration over the DFTFB during that time in order to conserve power. This means the controller must balance the short term penalty of a non-productive transition

State Category	Number of States	Number of Channels Produced	Average Power
SLEEP	2	0	5.36 μW
DCM	8	1	7.61 mW
DFTFB	1	8	17.92 mW
Trans. DFTFB	1	0	10.25 mW
Trans. DCM	1	0	10.25 mW

Table 2.1: Categories of processing states and their properties.

with the long term benefit of the presumably more favorable new state.

It can be seen from Table 2.1 that the number of channels affects the number of states, and thus, the size of the MDP state space. This has significant implications on the resources required to host an MDP-based control policy on the target system, and ultimately, on the scalability of this approach to channelizers with more than 8 channels. This concept will be explored in detail in Section 2.4.4.

2.4 MDP-Based Channelizer Control

In this section, an SLRF for modeling reconfigurable channelizers is developed with the goal of generating run-time control policies that can be steered in terms of multidimensional operational objectives, including latency, throughput, and energy efficiency. The procedure is to first create a Markovian model of the system, and then use an MDP solver to generate a control policy from the developed system model. It is important to note that the system and the environments that it operates in need not be Markovian or even stochastic in nature, and the Markovian assumptions are made as approximations expressly for the purpose of arriving at the control policy. These assumptions are validated by evaluating the resulting control policy on the real system (not the model) in its intended use case.

The resulting MDP-based dynamically reconfigurable channelizer is illustrated by the block diagram shown in Figure 2.5. The key feature of this system is that the channelization requests do not have direct control over the processing system. Rather, the channelization requests go only to the MDP-generated run-time control policy, which decides when and how to act on each specific request. The policy determines the best action to take, with the objective of maximizing the long-term average performance rather than solely based on an immediate reward. To make this determination, the policy uses models of the application and processing system characteristics. The policy may decide to reconfigure the processing system immediately if that is assessed as the best decision, or counter intuitively, it may decide to ignore a request that it predicts is a spurious request and would not justify a

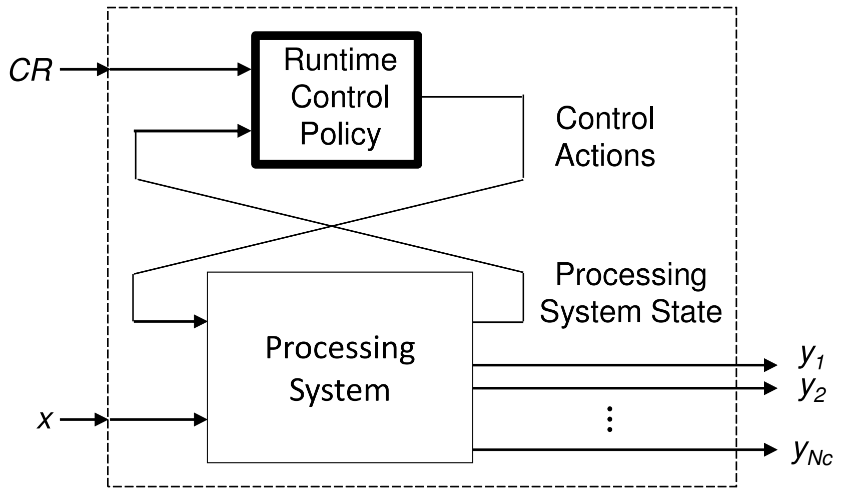

Figure 2.5: Dynamically reconfigurable channelizer.

reconfiguration event.

The key components of the MDP underlying the reconfigurable channelizer system are the 4-tuple ($\mathcal{S}$, $\mathcal{A}$, STMs, R), where the components of this 4-tuple are respectively referred to as the system *state space*, *action space*, *state transition matrices* (STMs), and *reward function.* In some contexts a scalar *discount factor* and a probability distribution for the starting state are also included in this list, sometimes making it a 5-tuple or 6-tuple. However, in this discussion the 4-tuple definition is sufficient.

The state space $\mathcal{S}$ is defined by enumerating all possible states of the external requests imposed on the processing system (channelization requests), as well as a list of modes that the processing system can be in at any time (reconfiguration states), which were detailed in Section 2.3. The combination (product) of these two subspaces (external requests and processing modes) yields the state space of the channelizer system.

For the action space $\mathcal{A}$, the MDP policy is given control over the reconfiguration decision, as well as selected parameter values within particular configurations. As a result, the action space consists of all the possible configurations and parameter values that can be commanded.

The *STM*s are a set of stochastic matrices that define the probability of the next state given the existing state, conditioned on a given action. There is one *STM* defined for each action. These matrices are obtained by multiplying together the independent statistics of the external channelization requests with the conditional statistics of the processing system's state transitions. The statistics of the channelization requests used to generate the *STM*s are given by the following equations.

$$P(CR_{j|i}) = \begin{cases} P_0(CR_j), & i = i_0 \\ \\ P_1(CR_j), & i \neq i_0 \end{cases}, \tag{2.2}$$

$$P_0(CR_j) = (p_{start})P_D(CR_j) + (1 - p_{start})1_{\{j=i_0\}} \tag{2.3}$$

$$P_1(CR_j) = (p_{stop})1_{\{j=i_0\}} + (1 - p_{stop})P_D(CR_j) \tag{2.4}$$

$$P_D(CR_j) = \beta^{\sigma(j)}(1 - \beta)^{N_C - \sigma(j)} \tag{2.5}$$

where i_0 is the state where no processing requests are incoming (representing periods of inactivity), $\sigma(j)$ represents the number of requested subchannels in the CR state j, β is a parameter used to simulate various levels of communication activity, and

p_{start}, p_{stop} are used to simulate the system entering and exiting periods of inactivity. The statistics of the processing system used to generate the STMs are detailed in Section 2.4.3.

2.4.1 Multiobjective Rewards

For the reward function R, the following methodology for incorporating multidimensional design objectives into an MDP-based channelizer design framework is used. Given a set $X = \{x_1, x_2, \ldots, x_{N_R}\}$ of N_R evaluation functions for key performance metrics, a reward function $R : (\mathcal{S} \times \mathcal{A}) \rightarrow \mathbb{R}$ is defined in terms of these metrics for each action in each state. Here, $\mathbb{R}$ denotes the set of real numbers.

Each evaluation function $x_i : (\mathcal{S} \times \mathcal{A}) \rightarrow \mathbb{R}$ is used to estimate system performance in terms of a specific implementation concern, such as average energy consumption, latency, or throughput. These estimation functions can be formulated at design time by using knowledge of the system and its available configurations, or measured at runtime by supporting instrumentation. The result of each evaluation function x_i is transformed by a mapping $g_i : \mathbb{R} \rightarrow [0, 1]$, which is defined at design time for each metric. These transformations are introduced to normalize the performance metrics in order to allow them to be combined into the single scalar output of R. This kind of transformation and combination follows the *scalarization* approach to multiobjective optimization, as described in [32].

The combination of the transformed results of the evaluation functions is performed through a set of weights $\rho = \{r_1, r_2, \ldots, r_{N_R}\}$, one corresponding to each metric, such that

$$(r_i \in [0,1] \text{ for each } i) \text{ and } (1 = \sum_{i=1}^{N_R} r_i). \tag{2.6}$$

Determining these weights ρ is a design time aspect of the SLRF. The weights are determined once and then continually used to steer any executions of the solver to seek policies that achieve the desired prioritization of metrics in consideration with the observed external environment statistics.

Once the evaluation functions X, transformations $\{g_i\}$, and combination weights ρ are determined, the reward function can be evaluated using Equation 2.7 for any given $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

$$R(s,a) = \sum_{i=1}^{N_R} r_i \, g_i(x_i(s,a)) \tag{2.7}$$

In the experiments, the rewards are defined as follows. First, g_1 is defined as the normalized rate of successful channelization requests. This can be expressed as $(\eta_r - \eta_d)/N_C$, where η_r represents the total number of channelization requests input to the system during a given time interval τ, and η_d represents the number of dropped requests (i.e., requests where there is a failure to produce the desired channel) during τ.

g_2 is defined based on a formulation in [28] for the *normalized power savings* of an electronic system. Specifically, in order to normalize power consumption and treat it as a form of savings, power consumption (x_2) is measured in each state and the minimum and maximum possible values are recorded. Then the power measurement are transformed relative to the maximum and minimum power that the system consumes in all of the possible states (g_2). The result is shown in Equation 2.8 and Equation 2.9.

$$g_2(x_2(s,a)) = \frac{x_{2,MAX} - x_2(s,a)}{x_{2,MAX} - x_{2,MIN}}, \tag{2.8}$$

where

$$\begin{aligned} x_2(s,a) &\equiv \mathit{Power\ Consumed}(s,a) \\ x_{2,MAX} &= \max_{s',a'}\{x_2(s',a')\} \\ x_{2,MIN} &= \min_{s',a'}\{x_2(s',a')\}. \end{aligned} \tag{2.9}$$

Note that this definition is consistent with the convention previously defined: the most power hungry state has $g_2 = 0$ (and thus is the least rewarded), while the least power hungry state has $g_2 = 1$ (and thus is the most rewarded).

The combination of rewards functions g_1 and g_2 effectively steer the MDP to find policies that are most productive at channelizing the incoming signal as per the channelization requests, while consuming as little power as possible on average.

2.4.2 MDP Solver and Policy

With the definitions and rewards described above, an off-the-shelf MDP solver can be employed to generate a policy that simultaneously seeks to maximize the rate of successful channelization requests while consuming the least energy possible, taking into account both the physical characteristics of the processing system as well as the independent statistics of the operating environment at the current time. In these experiments, the open source MATLAB solver MDPSOLVE [33] is used.

The resulting control policy has the form $f : \mathcal{S} \rightarrow \mathcal{A}$ — i.e., a mapping from states into actions. This mapping can be implemented as a function or simple lookup table that is invoked or accessed once per frame, respectively. To execute the controller, the incoming request is combined with the current processing system state. The result is then used as an index to lookup the operations involved in the next optimal control action.

In this example application, the total number of states is 3328 and the total number of actions is 13. For these quantities, the action can be encoded into 4 bits and thus 2 encoded actions can be packed into 1 byte of storage. The result is a policy that can be packed into 1.6kB. For the prototype hardware implementation, it was feasible to simply store the policy as a lookup table in RAM and index it to look up the next action.

2.4.3 Transition States

In this design context, the processing system is typically a deterministic, controllable machine, such as a general purpose processor (GPP), programmable digital signal processor (PDSP), field programmable gate array (FPGA) or graphics processing unit (GPU). The proposed framework assumes that this type of processing system can be modified or reconfigured through the action decision of the MDP. By definition, in MDP frameworks the system is assumed to transition probabilistically from one state to another as a result of an action decision. This abstract probabilistic transition viewpoint is not immediately amenable to modeling the transitions of a deterministic processing machine. Rather, the resulting state changes in the processing system are better described as a change that is guaranteed to occur but can take some fixed or variable amount of time to complete. Additionally, the change may take longer than a single frame to complete. Some examples of the types of operations typically encountered in this context that must be accounted for are: (1) computation of the schedule for a dataflow graph before being able to execute it, (2) allocation of memory from an operating system heap when initializing algorithms, (3) the block copy of code or data from a slower, larger long-term storage to a smaller, faster location (e.g., page fault), (4) the block copy of code from non-executable regions to executable regions (e.g., overlays), and (5) dynamic full or partial reconfiguration (DPR) of FPGA regions, to name a few.

To assign the required state transition probabilities in this context, suppose that the processing system receives action w in frame n while in state $sp^{(n)} = u$, and that this state/action pair is known to deterministically transition the processing system to a new state v in an amount of time denoted as $T_{u,v|w}$, which need not be an exact multiple of the frame period T_F.

If $T_{u,v|w} < T_F$, then the conditional State Transition Matrix for the processing system (SP_STM) is trivially computed by

$$SP\_STM_{i,j|w} = \begin{cases} 1, & j = v \\ 0, & otherwise \end{cases} \tag{2.10}$$

This represents a guaranteed (i.e., with probability 1) transition of the processing system to state v, that completes before the start of the next frame.

If, on the other hand, this transition takes longer than T_F, a new processing system state m is needed, which is defined as the state of being *in transition* from $sp = u$ to $sp = v$. In this case, the conditional SP_STM matrix is calculated by

$$SP\_STM_{i,j|w} = \begin{cases} 1, & i = j = v \\ 1, & i = u, j = m \\ 1 - c, & i = j = m \\ c, & i = m, j = v \\ 0 & otherwise \end{cases}, \tag{2.11}$$

where

$$c = \left[\mathit{floor} \left(\frac{T_{u,v|w}}{T_F} \right) \right]^{-1}. \tag{2.12}$$

For example, if the processing system transition takes 4.67 frames to complete and the action is held constant until the completion of the transition, then the system will begin transitioning immediately following the triggering action, and will remain in transition for 4.67 frames before arriving at the destination state. In this case, the conditional transition matrix states that with probability 1, the processing system will transition from the starting state to the transition state in the first frame, and then for each subsequent frame will remain in the transition state with probability 3/4, and will jump to the destination state with probability 1/4. This is exactly how the transition would appear to an agent who naively observes the processing state as a stochastic process during just the transition sequence. This agent would observe 3 non-transitions and 1 transition out of 4 trials.

The observations during the transition can be modeled as a Bernoulli random variable, as was done in [34] through the use of Bernoulli trials. Here, the two random outcomes are interpreted as those of remaining in transition and completing the transition. Then the Maximum Likelihood Estimator (MLE) of the Bernoulli parameter can be shown to be exactly as given by Equation 2.12. For this reason, the Bernoulli probability mass function is given by the corresponding row of the conditional transition matrix, as expressed in Equation 2.11. With knowledge (or an estimate) of the transition time from each state/action pair in the model, the entire set of SP_STM matrices can be populated in this manner.

2.4.4 Factorization

In this chapter, the MDP model and solver components are implemented and invoked at design time, in order to generate a control policy that is used at runtime. However, in order to implement runtime adaptation it is ultimately necessary to transfer the MDP model and solver to the target system such that the solver can be invoked periodically at run time. The solver can then be used to dynamically re-optimize the control policy in response to a changing external environment. Working towards this goal, in this section an analysis is performed of the required target platform resources necessary for embedded deployment of the MDP model and solver. The main aspects of resource utilization that are investigated here are (1) the size of the four MDP constructs ($\mathcal{S}$, $\mathcal{A}$, STMs, R) that need to be held in memory, and (2) the execution time of the MDP solver required to generate the control policy.

In this context, there are significant advantages to adopting the Factored MDP approach developed in [15]. In that work, knowledge of the stochastic interdependencies between the state space variables are exploited to reduce both the memory requirements and solver execution time.

In MDP problems, the state $s \in \mathcal{S}$ is constructed to model the problem the MDP is being applied to. Often this results in the state being an instantiation of a discrete multivariate random variable $\underline{Z} = (Z_1, Z_2, \ldots, Z_{N_Z})$, with each variable Z_i taking on values in $DOM(Z_i)$, where $DOM(X)$ represents the set of admissible values of the random variable X. A state is a set of instantiations of the N_Z random variables, and can be written as a vector $\underline{z} \in DOM(\underline{Z})$. The size of the state space is defined by the cardinality of this set, which is denoted as $|DOM(\underline{Z})|$. As a result, each row of each transition matrix for an MDP has width $|DOM(\underline{Z})|$, and describes the probability of reaching all possible combinations of the set of variables $(Z_1, Z_2, \ldots, Z_{N_Z})$.

MDPs with this kind of formulation are said to have a multivariate state space. When an MDP's STMs are stored in a structured way that uses knowledge of the causal relationships between these state variables to reduce storage size, the MDP is said to be *factored*. From the empirical observations of this application, it can be seen that this method can effectively reduce STM storage size considerably. However, it requires a specific conditional probability structure to be present in an MDP, and the data structures must be created by hand with specific knowledge of the exact structure.

This can require a subject matter expert in the loop anytime a transition probability changes, which can complicate runtime autonomous solving of an MDP that changes over time in unknown ways. In general, this requirement can be problematic if the underlying structure is not fully understood. The effectiveness of the technique is acknowledged here, but also the fact that it cannot always be used (for these reasons) is important as well.

Using the factorization approach, the state space of the channelizer can be represented as:

$$s = (CR_1, CR_2, \ldots, CR_{N_C}, CF_1, CF_2). \tag{2.13}$$

Here, CR_i is the channelization request for channel i, CF_1 is the top-level processing configuration, and CF_2 is the processing subconfiguration. The benefit of using this scheme is that it enables the explicit specification of the stochastic interdependencies of the variables within the state space. With this in mind, factored MDPs make use of Dynamic Bayesian Network (DBN) diagrams [35] to explicitly define and illustrate these dependencies.

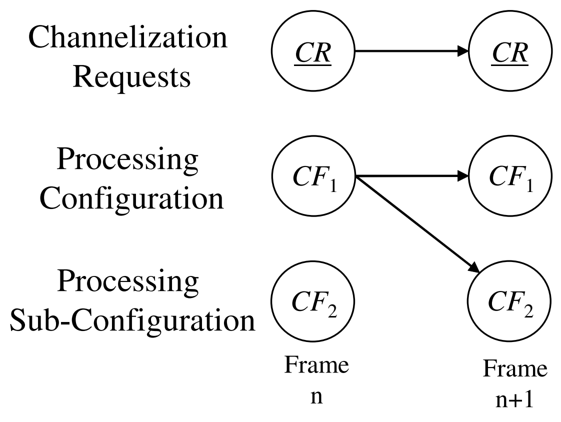

Figure 2.6: Dynamic Bayesian network representation of the channelizer state space.

A DBN diagram of the channelizer's STMs when conditioned on an MDP action is shown in Figure 2.6. Note that the $(CR_1, CR_2, \ldots, CR_{N_C})$ requests are grouped together into a single vector $\underline{CR}$ for conciseness. A stochastic dependency between two variables in the state space (from one time frame to the next) is denoted via the presence of an arrow between the dependent variables. The absence of an arrow denotes independence. Thus, the diagram shows that the joint probability distribution of the channelization requests is dependent only on the requests in the previous frame, and is independent of the processing configuration. The processing configuration is dependent only on the previous processing configuration (since reconfigurations are not instantaneous). However, this dependency is only on the top-level processing configuration (e.g., DCM, DFTFB, etc.) and not on the subconfiguration (e.g., the filter coefficients).

Knowledge of this underlying stochastic structure within the state space allows for considerable reduction of the size of the data structures required to store the MDP model. The effect on the largest of these components (the STMs) is highlighted. Only the conditional probabilities with respect to the dependent variables need to be stored, rather than with respect to all variables — as would be necessary in an equally sized state space where the underlying stochastic structure is unknown. The factorization made possible by the knowledge is represented in Equation 2.14, where the superscript n is used to denote the time index. The rearrangements are made possible through (1) independence between the channelization request and processing configuration, and (2) independence between the channelization request and the MDP action.

$$\begin{aligned} p(s^{(n+1)}|s^{(n)}, a^{(n)}) &= p(\underline{cr}^{(n+1)}, {cf_1}^{(n+1)}, {cf_2}^{(n+1)} \mid \underline{cr}^{(n)}, cf_1^{(n)}, {cf_2}^{(n)}, a^{(n)}) \\ &= p(\underline{cr}^{(n+1)} \mid \underline{cr}^{(n)})\, p({cf_1}^{(n+1)}, {cf_2}^{(n+1)} \mid {cf_1}^{(n)}, a^{(n)}) \end{aligned} \tag{2.14}$$

The resulting reduction in the number of elements in the STMs is shown in Equation 2.15. This reduction represents a significant savings. Note that the quantity shown in Equation 2.15 is the cardinality of the sets, which is a count of the number of elements regardless of what underlying data type is used for representation in the MDP model and solver algorithms. For example, if the data type used is a 16-bit or 32-bit representation, the total storage size would be 2 bytes or 4 bytes per element, respectively.

$$\begin{aligned} |\mathcal{S}|^2\ |\mathcal{A}| \gg |\mathrm{DOM}(\underline{CR})|^2 + \\ |\mathrm{DOM}(CF_1, CF_2)|\ |\mathrm{DOM}(CF_1)|\ |\mathcal{A}| \\ 121.8\text{x}10^6 \gg 66.3\text{x}10^3 \end{aligned} \tag{2.15}$$

2.4.5 Stability

Once an MDP-generated policy is used to control the processing system, the stability of the resulting dynamics can be analyzed. There are two aspects of stability that arise in the context of this SLRF.

First, it is important to design the model of the processing system such that for a given action, the model is guaranteed to eventually transition to the desired processing state (after any transition delays and transient dynamics have settled) and to remain in that state indefinitely for as long as that action is held constant. For example, the action that configures the channelizer to be in the DFTFB state should eventually result in it reaching that state, and it should remain in that state indefinitely until another action is selected. This characteristic of the model aids stability and can be defined formally in the context of Markov chains, which allows it to be verified systematically at design time before attempting to generate the optimal policy for the MDP. Such a step of systematic model checking can aid in larger modeling efforts where the criteria may not be immediately obvious, as in the case of complicated transition dynamics including multiple transition states.

The product of terms format of Equation 2.14 that results from factorization allows for the transition probabilities of the processing system to be specified separately from those of the external environment. This subset of the full STMs can be viewed as a smaller set of state transition matrices $\mathcal{M}_{CF}$ for the processing system only, not the external environment. For a given action, the processing system is modeled to transition according to this state transition matrix and thus forms a Markov chain with state vector $\underline{CF}$. The stability criterion described above is equivalent to requiring that each transition matrix in $\mathcal{M}_{CF}$ qualify as an *absorbing* Markov chain as defined in [36], and additionally that only one absorbing state exists per action. Such a Markov chain is one where each state can reach the absorbing state, and once the absorbing state is entered it cannot be exited. In this SLRF, the way to leave the absorbing state is to change the action, which effectively changes the processing system transition probabilities and creates a new Markov chain that allows for transition out of the state. This absorbing characteristic can be inspected numerically in the model, by verifying that all of the $\mathcal{M}_{CF}$ matrices conform to these numerical restrictions.

The second aspect of stability that arises is that of the dynamics of the full MDP, which includes both the processing system and the external environment. The dynamics of the external environment are exogenous and uncontrollable, and thus not a concern from a stability viewpoint. However, the environment affects what the MDP-generated policy will contain, and by extension the resulting sequences of actuations that the MDP-generated policy exposes the processing system to. Given that the processing configurations are a finite set, instability could materialize as an excessive reconfiguration back and forth between processing states that negatively affects the overall closed loop performance with regard to the defined performance metrics. The presence of such instabilities can be detected using the simulations introduced in Section 2.5.

In the case of this form of instability, the designer can correct the dynamics by adjusting the reward weights ρ introduced in Section 2.4.1. A thorough analysis of why some reward functions lead to this form of behavior can be found in [37].

2.5 Results

To evaluate the effectiveness of the *MDP-based Reconfigurable Channelizer System* (*MRCS*), a simulation was developed with external requests that follow the statistics of the two use cases — here termed *IID* for the i.i.d. requests of Use Case A (introduced in Section 2.3.1), and *SEQ* for the sequential sensing of Use Case B (introduced in Section 2.3.2). In the following sections three evaluations are performed. First, the results are compared against those of manually generated policies, that are considered representative of a typical approach used in industry at the time of this writing. Second, the results are compared against another method published by researchers. Third, the effectiveness and trade-offs associated with modeling transition states are explored.

2.5.1 Comparison with Manually Generated Policies

In order to evaluate the effectiveness of the MDP generated control policy, several alternative control policies are created to compare it against. These are referred to as the "manually generated" policies, and contrasted with the set of "MDP generated" control policies. The manually generated policies were generated through intuitive heuristics, by first defining common sense rules for controlling the system in question, and then translating those rules into code. This represents the traditional method that an embedded software developer would use to create a reconfiguration policy. For the manually generated alternatives, the rules and resulting policies are as follows:

1. DFTFB — This policy keeps the DFTFB algorithm on the chip at all times, and invokes it in all frames regardless of the external requests. This policy was used purely as a starting baseline, as this policy represents the absence of reconfiguration options, using the most productive and processor intensive channelizer available in the system at all times to meet all requests.

2. DFTFB+Sleep — This policy also keeps the DFTFB algorithm on the chip at all times. However, if the number of requested channels is 0, the DFTFB is put into sleep mode. Otherwise, the DFTFB is kept on.

3. DCM+Sleep — This policy keeps the DCM algorithm on the chip at all times. If the number of requested channels is 0, the DCM is put into sleep mode. Otherwise, the DCM is kept on and applied to produce one of the requested channels.

4. DFTFB+DCM+Sleep — This is a set of policies that use both the DFTFB and DCM algorithms. The reconfiguration decision occurs based on how many channels are requested in the upcoming frame. If less than DFT_THRESH channels are requested, the DCM algorithm is used. If more than this threshold are requested, the DFTFB algorithm is used. Additionally, if the number of requested channels is 0, the algorithm that is currently is loaded is put into sleep mode. If a reconfiguration is in progress, it is allowed to finish regardless of incoming requests. The DFT_THRESH parameter is varied from 2 to 6, resulting in 5 different control policies.

In order to compare the policies objectively, the following experimental setup was created on the EFM32GG development board. Both channelizer algorithms were implemented in C, compiled and stored on the system's non-volatile storage. A MATLAB simulation was created that produced a time series of channelization requests having the statistics described in the two use cases A and B. The time series output of the simulation was translated to a C array and stored on the EFM32GG. A test harness was written on the EFM32GG, which was driven by a periodic timer interrupt. At the interrupt rate, the next channelization request was pulled from the stored array and that channelization request was then used as an input to the dynamically reconfigurable channelizer system.

This system was implemented in C and executed on the EFM32GG. In order to facilitate an objective comparison of control policies, all of the manually generated policies were stored as Lookup Tables (LUTs) in addition to the MDP generated policies. This allowed both the manually- and MDP-generated policies to be invoked by suitably swapping out the contents of the LUT.

As part of the test harness, a small amount of diagnostic code was incorporated to compute performance objective 1 (channelization productivity) in real-time. This computation was performed by comparing the produced channelizer outputs with the requests. A channelization request that was successfully carried out was labeled a success. Conversely, a request that was not carried out was labeled a failure (e.g., if the processing system was in a reconfiguration state during a frame with channelization requests in it, or if a configuration was in place that could not produce enough output channels, etc.). The ratio of the successful outcomes to the number of requests was used to compute a success rate, which was used as a measure of system productivity. The measured productivity results were periodically streamed to a laptop computer using the ARM on-chip Real-Time Trace (RTT) functionality, and EFM32GG Single Wire Output (SWO) port. The streamed output for each case was tabulated and used for comparison.

Metric 2 (CPU power consumption) was measured by using the EFM32GG board's energy monitoring tools. These development tools allowed a very accurate current measurement to be taken, showing the exact current drawn by the CPU over time for each control policy. The total current drawn over the total simulation time was used to create a single metric for average power consumption. Thus, a highly repeatable experimental setup was applied, where all experimental settings were kept the same from case to case with the only difference being the control policy being used.

Results of these experiments are summarized in Figure 2.7. Here, each point in the figure represents the average performance of one policy over the entire simulation. The MDP policies generated by different values of r_1 are connected together, illustrating a Pareto front generated by the suite of MDP policies. The manually generated policies are plotted without any connecting lines. If the distance from the origin is used as a scalar metric of performance, the MDP generated policies all outperform or perform equally to the best manually generated policies.

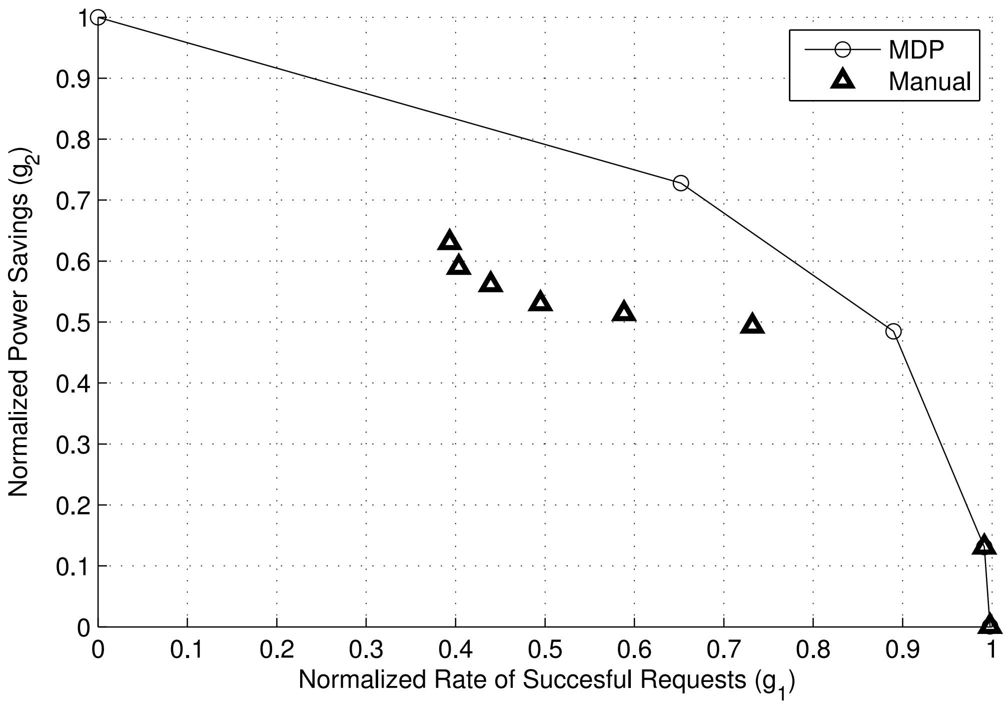

Figure 2.7: Policy comparison results.

2.5.2 Comparison with mHARP

Next, the *MRCS* is compared to a competing published method, the Highly Adaptive Reconfiguration Platform (HARP), introduced in [3]. One modification was needed, as the published HARP made decisions purely to optimize energy efficiency. This was inadequate for the channelizer case study, as the most energy-efficient result is one where the system never leaves its sleep state. To remedy this, the single metric in HARP was replaced with the multidimensional reward formulation in Section 2.4.1, to construct a useful HARP policy and also to provide a fair comparison between the two methods. This modified method is referred to here as *multiobjective HARP* (*mHARP*).

For each of the two competing techniques, 10 scenarios were created by varying the Bernoulli parameter in use case A, and another 10 by varying the channel dwell time in use case B. The result is 20 simulations where the proposed method and the baseline method (described below) were allowed to implement and run the optimal control policy for the given use case and external environment. The system characteristics and measurements described in the previous section were used to define the processing system under control. The results from these experiments are summarized in Figures 2.8 and 2.9, for use cases A and B, respectively. As previously mentioned, HARP requires a priori tuning for a given desired system dynamic. In this simulation, mHARP was optimized for power savings. The results show that when tuned in this way, mHARP does well in this metric for all scenarios (producing slightly better performance than the MRCS approach), but greatly sacrifices performance in the success rate for half of the scenarios. Conversely, when mHARP was optimized for the success rate, large shortcomings in the power savings were observed. In contrast, the MRCS *involves no a priori tuning, and optimizes all decision making for each scenario individually* without compromises. These results show the MRCS to have greater robustness to a wide range of parameters in different applications, all without any human-in-the-loop intervention.

2.5.3 Trade-offs in Modeling Transition States

An analysis was performed into the effectiveness of modeling processing state transitions, as described in Section 2.4.3. Although the prototype system did not

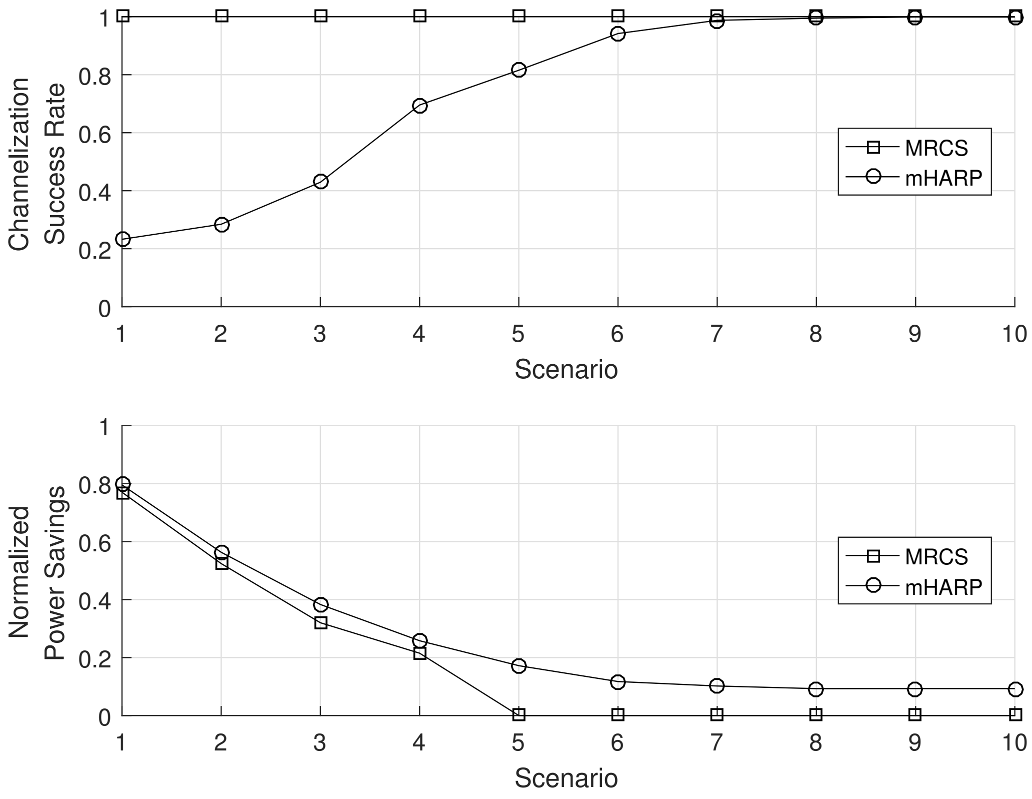

Figure 2.8: Experimental comparison between MRCS and mHARP, for IID use case.

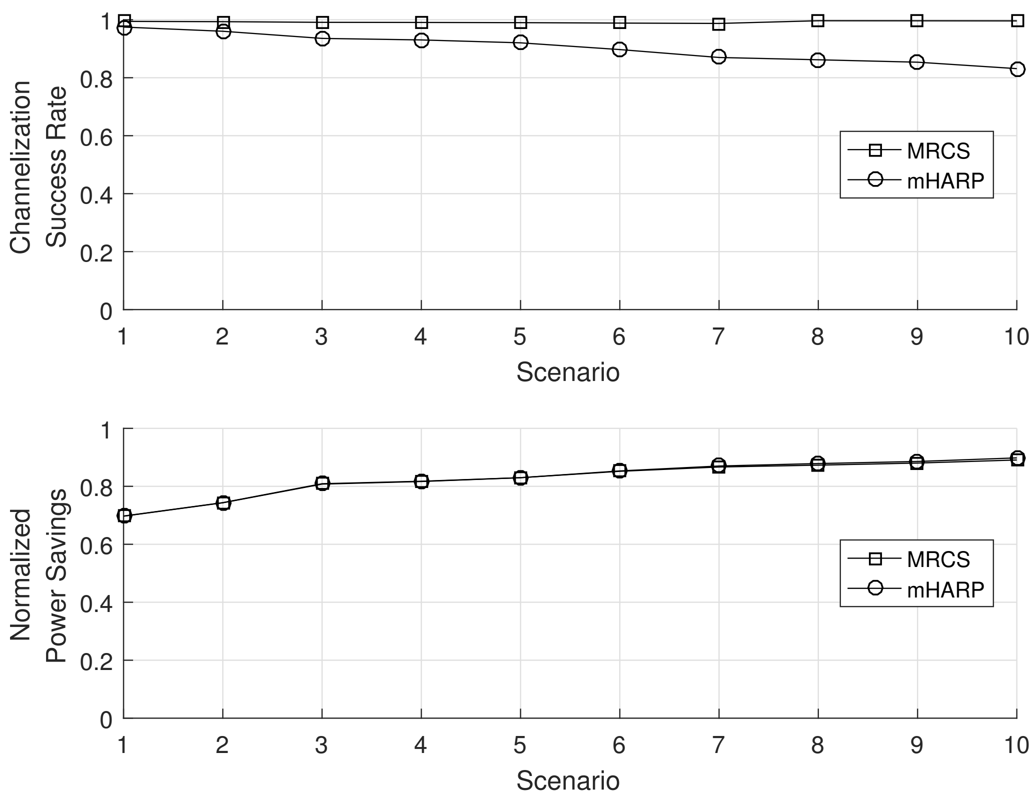

Figure 2.9: Experimental comparison between MRCS and mHARP, for SEQ use case.

incur large reconfiguration delays, larger delays are anticipated in future work as it scales up to larger channelizer applications. Adding transition states to the MDP model has the undesirable effect of increasing the size of the state space, which is known to increase the size of the model's data structures as well as the execution time of the policy generation algorithms. In order to make informed modeling decisions, it is crucial to understand what is gained at the expense of these costs. With these goals in mind, one of the scenarios of the *IID* application was selected for exploration, and modified in two ways.

First, the dynamics of the processing system were modified by changing the amount of time that transitions of the top-level reconfigurations would take to complete. This delay was varied between 1 and 5 frames, representative of a range of a small reconfiguration delay to a large delay. Second, two alternative MDP modeling approaches were used and compared: one with the transition states modeled and one without.

Delays Modeled	STM Size [Elements]	Solver Execution Time [Seconds]
No	66020	17.2
Yes	66394	24.0

Table 2.2: Modeling costs with and without transition delay modeling.

The cost of the additional modeling is shown in Table 2.2. The increase in the size of the STMs is practically negligible, however the increase the solver's execution time is not. The benefits of this more expensive modeling come at run-time, and

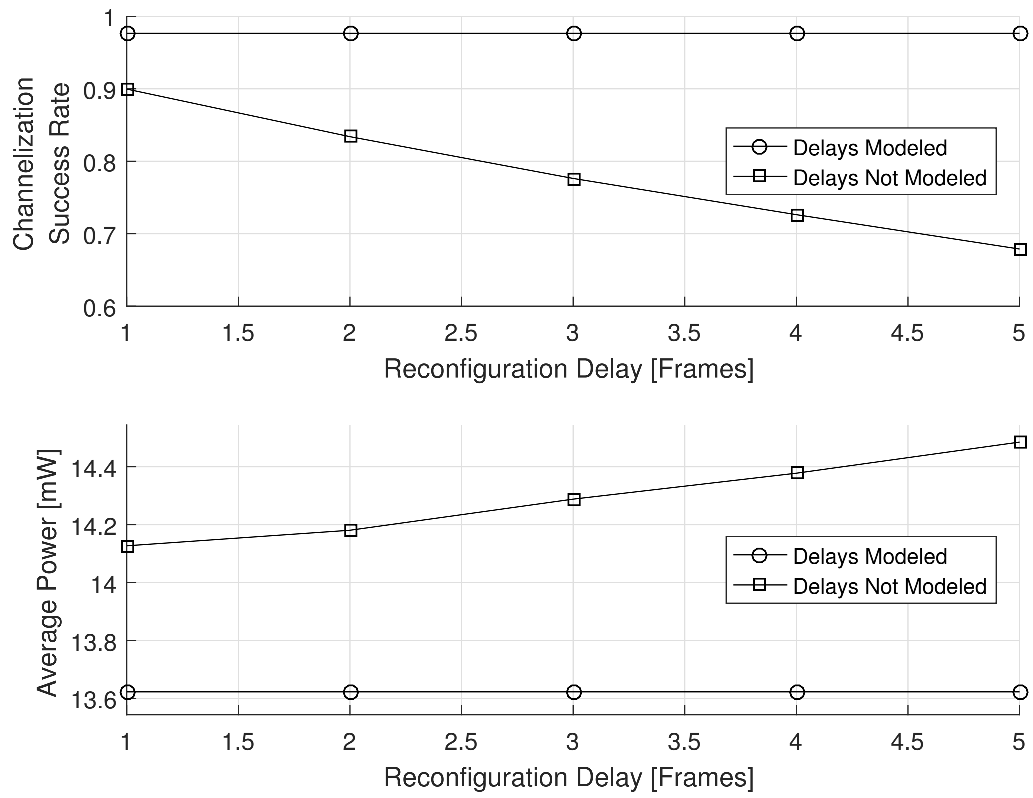

Figure 2.10: Run-time performance with and without transition delay modeling.

are shown in Figure 2.10. This figure shows the resulting assessment in terms of the performance metrics defined in the previous section.

From this assessment, it can be seen that when transitions are not modeled, the performance of the system (with respect to both metrics) degrades proportionally with the length of the reconfiguration delays. This degradation is attributed to the system spending more time in a non-productive reconfiguration state. In comparison, the MDP that has the transitions modeled does not exhibit this performance degradation. These results are attributed to the fact that the MDP with transition states is able to consider the reconfiguration penalties in its decision criteria, and as a result is more “reluctant” to trigger costly reconfigurations.

2.6 Conclusions and Future Work

In this chapter, a methodology was presented for design and implementation of adaptive digital channelizer systems, and a novel channelizer design was demonstrated, called the MDP-based reconfigurable channelizer system (MRCS), that is derived using the new methodology. This methodology and the MRCS employ compact, system-level models based on MDPs to generate control policies that optimize the required embedded signal processing tasks in terms of relevant, multidimensional design optimization metrics.

The MRCS was designed using a System-Level Reconfiguration Framework (SLRF), which provides a systematic methodology for dynamic adaptation of embedded signal processing configurations. The framework includes multiobjective optimization at its core, embracing the multifaceted nature of embedded systems design, where making strategic trade-offs among conflicting goals is critical. As a result of this emphasis, the framework can jointly optimize power consumption and real-time throughput, and can readily be adapted to address other combinations of metrics that are important for a given application.

The effectiveness of the method was shown in simulation, using empirical measurements for the properties of an experimental signal processing system. Through extensive simulations, it was shown that the MRCS outperforms the prior state-of-the-art in terms of robustness to changing applications and scenarios.

Useful directions for future work include adapting the MDP-based, reconfigurable channelizer design methodology to derive dynamically reconfigurable forms of other types or other combinations of channelizer architectures, and continuing to generalize the proposed design methodology to address broader classes of embedded signal processing applications.

One requirement of the proposed SLRF is that the statistics of the external environment and reconfiguration dynamics must be known at design time. In certain applications, this may not be feasible, or they may be time-varying to such a point that a policy generated offline at design time may experience a reduction in effectiveness as these properties change.

An important complement to this framework is in learning strategies to estimate these statistics at runtime for systems where they are not constant or not known up front. These running estimates can then be used to periodically re-optimize the control policy and keep it performing optimally across time-varying use cases and a time-varying environment. These concepts are explored in the chapters that follow.

Chapter 3

Efficient Solving of Markov Decision Processes on GPUs using Parallelized Sparse Matrices

3.1 Introduction

This chapter presents a novel algorithm that enables fast and efficient use of MDPs in real-time on resource constrained signal processing systems that are equipped with graphics processing units (GPUs). Material in this chapter was published in preliminary form in [38].

For many years, researchers have been applying MDPs in limited ways to control computing systems at runtime [34]. Typical application domains that have applied MDPs in this way are artificial intelligence [35], multirate digital signal processing [13], and wireless sensor networks [39], among many others.

MDPs are often regarded as useful tools in theory. However, they have often been deemed too computationally demanding to be fully leveraged in resource constrained computing systems due to the processing time and RAM required to do so [16]. This suggests that their full potential and utility in this class of systems has not yet been reached.

In this chapter, recent advancements in parallel processing made possible by embedded GPUs are applied to help bridge this gap. The benefits of GPUs in accelerating many important types of computations is now well accepted in the embedded systems community [40, 41]. The application of GPUs to accelerate MDP algorithms has been reported on in recent published work (e.g., see [42]). However, this previous work focuses on one MDP use case, and does not include analysis on how acceleration can be optimized for different kinds of MDPs. One important contribution of this chapter is to provide a general analysis of GPU-based MDP acceleration. In particular, an analysis is presented detailing how much parallelism exists and where it can be found, as well as a parameterized blueprint for how many parallel threads and GPU kernel executions can be invoked as a function of the MDP dimensions and parameters.

Additionally, acceleration results are produced that are significantly beyond other MDP techniques that have used GPUs. These improvements are established by incorporating recent advancements in sparse linear algebra on GPUs [43]. By integrating GPU acceleration and sparse linear algebra techniques to MDP solver design, a novel algorithm is developed, called Sparse Parallel Value Iteration (SPVI). SPVI enables fast, memory-efficient implementation of MDP solvers on resource constrained GPU platforms, such as mobile and embedded GPU SoCs.

With these capabilities, SPVI enables the use of MDPs in new and more useful ways than was previously possible. Also, SPVI relaxes constraints and limitations that other MDP techniques have imposed, which can further facilitate more widespread use of MDPs in signal processing systems. The source code to SPVI is included in the software package described in Chapter 5.

The remaining sections of this chapter are organized as follows. Section 3.2 contains a survey of the literature, and provides history and background for the chapter. In Section 3.3, SPVI is defined in detail and in Section 3.4, applications are explored to illustrate how the concepts that SPVI uses materialize in real-world use across typical use cases. In Section 3.5, the SPVI algorithm is compared experimentally against the prior state-of-the-art in MDP solver algorithms, and shown to have considerable advantages with respect to performance and feasibility of implementation on modern embedded computing hardware.

3.2 Background and Related Work

The different ways in which an MDP can be applied to control a computing system are vast and varied. MDPs provide a generic decision making framework that uses abstract concepts including *states*, *actions*, *transition probabilities* and *rewards*. Once these concepts are defined they are then passed to an MDP solver, which is an algorithm that produces an optimal policy with respect to those definitions. The policy is a mapping from states to actions, such that an agent using the policy looks up what action to take for any given state.

However, there is no consensus in the literature regarding exactly how to map elements of computing systems to components in the MDP framework. This mapping is in general left to the designer who is applying the MDP to solve a specific computing problem. For example, a processing system can be commanded to run a particular algorithm (and this can be modeled as a state), or that same command can be modeled as an action instead, or it could be modeled as both (an action that leads to a state). Also, the choice of granularity for these definitions is important — e.g., are two invocations of the same algorithm with a slightly different parameter value considered two different actions, or the same action?

There are several approaches in the literature to map elements of computing systems to MDP states and actions, and these different approaches lead to different results, with implications in both the final policy performance as well as how hard it is to model and solve the MDP. One of the earliest known applications of using an MDP to control resources in computing systems at runtime is [34]. Notable examples of differing approaches include the reconfigurable digital filter presented in Chapter 2, a reconfigurable router [28], a power management module for a microprocessor [44], a smartphone scheduling program that synchronizes email efficiently [45], and a collision avoidance algorithm for commercial aircraft [46].

3.2.1 MDP Solvers

One of the first challenges associated with using MDPs is choosing what constitutes a state, an action and a reward. After that is decided, the associated MDP data structures must be stored on a computer and used as the inputs to the MDP solver to produce a policy. With this policy, the runtime decision framework consists of observing what state the system is in, and using that as input to the policy to determine what action to take.

The classical methods to solve MDPs are algorithms known as Value Iteration, Policy Iteration and Modified Policy Iteration [35]. All of these algorithms produce an optimal solution to the MDP problem, with different approaches leading to different implications in the execution time, power requirements and memory use of the solver routines.

These classical MDP solver algorithms suffer from the same issue as most systems that try to reason using computations of probability distributions: the framework's data structures grow exponentially with the size of the state space. A large state space is desirable in order to have sufficient model expressiveness to tackle difficult decision problems, but this desire is at odds with the resource requirements needed to solve an MDP that has a large state space. The upper limits on memory consumption that are available on typical embedded computing systems can often easily be reached, before many important system details have been modeled.

More specifically, the total size of (number of elements in) an MDP's State Transition Matrices (STMs) are $N_S^2 N_A$, where N_S and N_A are the number of elements in the state space and action space, respectively. The STMs are the largest data structures in the MDP, and usually the most difficult structures to store and process, due to their large size. The STMs are large matrices even for modest choices of N_S and N_A, and if one were to add a state variable with L states to the state space, this addition would increase the size of the STMs by L^2. Besides the storage space and memory requirements to store large data structures, increasing the state space also causes the solver's execution time and power consumption to grow exponentially as well.

Thus, for computing systems that operate under strict resource constraints, it is not enough to frame an MDP in a way that produces a well performing solution. There is also the practical issue of whether the solver can be successfully implemented on the targeted platform, and whether it can complete in an amount of time reasonable for the application.

This so-called *curse of dimensionality* [16] usually results in limiting the use of MDPs to a mode of deployment that greatly hampers their usefulness: the solver is invoked only once offline, and then the generated MDP policy only (not the entire framework required to solve the MDP) is used on the target system. This scenario is suboptimal and limiting if the problem inputs are unpredictable, constantly changing, or dependent on the environment. To overcome these limitations, one approach is to design efficient and *compact* MDP solvers, which enable the full MDP to be stored and solved on-demand on the target system. Such *embedded MDP* deployment leads to a more intelligent and adaptive class of embedded systems, which can learn, adapt and autonomously re-optimize themselves for changing conditions and use cases.

The design of compact solvers has been the goal of various researchers in recent years. For example, Boutilier et al. propose *factored* MDPs as a method for compact representation of large, structured MDPs [15]. This method can work quite well in principle. However, it requires a specific conditional probability structure to be present in an MDP, and the data structures must be created by hand with specific knowledge of the exact structure. The state transition matrices must be manually converted into tree-shaped conditional probability structures. This can be difficult, time-consuming and typically requires a subject matter expert in the loop anytime a transition probability changes, which effectively prevents or at least greatly complicates runtime autonomous solving of an MDP that changes over time. Additionally, this requirement can be problematic if the underlying structure is not fully understood. In contrast, the SPVI algorithm has no such limitation. SPVI does not require having any knowledge of the structure of the probabilities in the STMs, nor does it require hand-crafting of the data structures into tree-shaped objects.

Hoey et al. detail an algorithm similar to Value Iteration using Algebraic Decision Diagrams [47]. This approach shows good results in taming the curse of dimensionality, but imposes the same restrictions as [15] and thus has the same limitations.

In another example, Jonsson and Barto present an algorithm that performs hierarchical decomposition of factored MDPs into smaller subtasks to help alleviate the growth in complexity [48]. This approach can be effective, but also requires a priori knowledge of the causal structure within the MDP. This knowledge can be very difficult or impossible to know for many MDPs, a shortcoming identified by the authors themselves. Also, this method requires that the MDP have this decomposability property, which is not always the case.

In a similar spirit, Lin and Dean [49] present a method to solve a large MDP by first decomposing the state space into regions, determining actions to take within those regions, and then using novel approaches to combine the resulting sub-policies into an overarching policy that solves the original, large MDP. The authors note in this work that the decomposition must be done a priori by a domain expert. This decomposition is not guaranteed to be feasible, and when it is feasible, it can be very difficult to perform.

In contrast to all of this prior work, the proposed SPVI approach requires no special knowledge of the structure within an MDP in order to be used.

3.2.1.1 POMDPs and Approximate Solvers

Another critical issue in deploying MDPs into embedded systems is that the runtime system may not have any way to know exactly what state it is in. In this case, the problem statement changes to that of a Partially Observable MDP (POMDP). Observability here refers to the ability of the system to observe its place in the state space.

In POMDPs, the policy is no longer a mapping from discrete states to discrete actions, but instead a mapping from a continuous space known as the belief vector, to a discrete action [35]. The belief vector represents a probabilistic interpretation of the system's best guess for what state it is likely in. Since the discrete state space is a subset of the continuous belief vector, POMDPs carry with them even more computational burdens for two reasons: 1) the solver is tasked with solving a harder problem, and 2) the system invoking the policy must maintain an evolving time series of the current belief vector, in order to use it as the input to the policy.

The Artificial Intelligence community has spent considerable effort developing computationally efficient solvers for POMDPs. A good survey for work in this area can be found in [50]. Many of these solvers are approximate solvers (not exact solvers), in that they seek to reduce computation by settling for finding sub-optimal solutions that may be very close to the exact solution.

POMDP solvers can be used to solve (fully observable) MDPs, but not vice-versa. Although this chapter focuses on solving MDPs and not POMDPs, it is worthwhile to investigate whether recent advances in approximate solutions to POMDPs can help in solving MDPs efficiently. In Section 3.5, findings of this survey are utilized to obtain and benchmark some contenders, and use these as candidates in comparisons for completeness.

3.2.2 Sparsity

A novel feature of the SPVI algorithm is the exploitation of sparsity in in the MDP data structures. In this context, *sparsity* is defined as the percentage of zero-valued elements in the MDP *STM*s. Wijs et al. [51] present a promising method to decompose MDPs into subgraphs, exploiting sparsity on GPUs. However, the method is presented in the context of model checking for formal methods in software engineering. More research is required to incorporate this method into an MDP solver.

A caveat must be added to this approach, in that high sparsity is not guaranteed to exist in an MDP. In other words, one can easily synthesize an artificial MDP with its contents populated with random numbers that meet the definition of a valid MDP. Such a synthetic MDP can be generated in a way that does not exhibit high sparsity. However, when MDPs are constructed to solve real-world problems, the resulting data structures are often highly sparse. The proposed SPVI approach can be applied to any MDP (sparse or not). However, the largest performance increases will result only if the MDP is sparse.

Section 3.4 elaborates the claim that sparsity is commonly found in MDPs, and provides both conceptual and empirical evidence to support it. The aim is to show that the assumption of sparse *STM*s is not a major limiting factor for practical MDPs.

3.3 Method

As mentioned in section 3.2.1, there are three well known algorithms for solving MDPs. This chapter focuses strictly on Value Iteration and leaves exploration of the other two algorithms as an interesting direction for future research. First, the characteristics of the Value Iteration algorithm that make it suitable for acceleration via parallel processing are detailed. Then, the focus turns to the important concept of sparsity in MDPs. Finally, SPVI is presented in detail. SPVI can be viewed as a version of Value Iteration that uses both parallel processing and sparse matrix representations.

3.3.1 Parallelization

Value Iteration is an algorithm that is used to generate an optimal policy for an MDP. In Value Iteration, a real number (or value) $V(s)$ is associated with each state s. This mapping is known as the Value Function. The value $V(s)$ represents the expected reward that can be obtained from state s. The Value Function V is derived by using the iterative procedure shown in Equation 3.1, which starts out assigning a value of zero for each state and then incrementally converges from that to the optimal Value Function. Once sufficient iterations are performed, the optimal Value Function is known and the optimal MDP policy can be obtained trivially from it. This process of deriving the Value Function is a form of dynamic programming [52].

$$
\begin{aligned}
&V^0(s_i) = 0 \\
&V^n(s_i) = \max_{a \in \mathcal{A}} \{ R(s_i, a) + \beta \sum_{s_j \in \mathcal{S}} [P(s_j | s_i, a) V^{n-1}(s_j)] \}
\end{aligned}
\tag{3.1}
$$

In Equation 3.1, $V^n(s_i)$ is the approximation to the Value Function in state s_i at loop iteration n, $\mathcal{S}$ is the discrete state space, $\mathcal{A}$ is the discrete action space, $R(s_i, a)$ is the reward function for each state-action pair (s_i, a), β is a scalar discount factor, and $P(s_j|s_i, a)$ is the probability of transitioning from state s_i to state s_j after taking action a. Arranging the conditional probabilities in a matrix with s_i as rows and s_j as columns gives the State Transition Matrix (STM) for action a.

A good discussion of strategies for computing the stopping criteria used to terminate the iteration in Equation 3.1 can be found in [15]. Under this optimal solution, the MDP policy contains the optimal action to take in a given state to maximize the expected reward. It is worthwhile to note that although the iteration is initialized here using the zero vector, it is shown in [53] that the iteration converges for any initialization vector.

In SPVI, the capability of modern embedded GPUs to compute using massive parallelization is utilized. The case studies in Section 3.4 and performance benchmarks in Section 3.5 use a device in the NVIDIA Jetson family of embedded GPUs. These processors are small, power-optimized General Purpose GPUs (GPGPUs) that can be used in embedded systems to enable hundreds or thousands of parallel threads of execution in highly Size, Weight and Power (SWaP) constrained systems. To describe the acceleration of Value Iteration through parallel processing, the following section contains an analysis of how much parallelism can be exploited in Equation 3.1.

There are three components of Equation 3.1 that can be accelerated with parallel execution of threads, as long as two copies of the Value Function are maintained in memory (one for V^n and one for V^{n-1}). First, the elements inside the $max\{\}$ operation can be computed independently for each action $a \in \mathcal{A}$. Second, each iteration for the entire second line of Equation 3.1 can be computed independently for each state $s_i \in \mathcal{S}$. Third, the stopping criteria typically used is the comparison of $\|\underline{V}^n - \underline{V}^{n-1}\|_\infty$ to a scalar threshold, and this norm operation can be computed using a parallelized dimensionality reduction kernel.

With these characteristics, the Value Iteration algorithm is inherently well suited to significant acceleration through the high levels of parallelization achievable with embedded GPUs.

3.3.2 Sparsity

The previous section described the Value Iteration algorithm as it would be implemented on a single-threaded CPU. If sufficient parallel processing resources are available, it is possible to achieve acceleration through computation of independent operations simultaneously. With this goal in mind, Equation 3.1 is rewritten into Equation 3.2, using matrix and vector representations to move away from the sequential-looping approach:

$$\underline{V}^n = \max_{a \in \mathcal{A}} \{ \underline{R} + \beta \cdot \boldsymbol{M} \cdot \underline{V}^{n-1} \}. \tag{3.2}$$

Here, $\underline{R}$ represents the reward function for each state and action flattened into a length $N_S N_A$ column vector, where N_S and N_A are the number of elements in the state space and action space, respectively; β remains a scalar; and $\boldsymbol{M}$ represents the vertical concatenation of all N_A of the $N_S \times N_S$ transition matrices into an $(N_S N_A) \times N_S$ matrix.

Through execution profiling, it was consistently observed that a large portion of the total computation time in Value Iteration is spent multiplying the $\underline{V}^{n-1}$ values by the transition probabilities. In other words, a large portion of the computation time in Equation 3.1 is spent performing the summation loop over $s_j \in \mathcal{S}$, which needs to be repeated $(N_S N_A)$ times for each iteration. Equivalently, in Equation 3.2, the majority of the time is spent performing the large matrix-vector multiplication $\boldsymbol{M} \cdot \underline{V}^{n-1}$. This trend was observed consistently on single threaded CPU implementations using profiling timestamps, as well as on parallelized GPU implementations using the NVIDIA Profiler nvprof.

Under the assumption that the matrix $\boldsymbol{M}$ is sparse, it follows that that the majority of the computation time in Value Iteration solvers is spent multiplying a large sparse matrix by a vector. In other words, this time is spent multiplying elements by zero and then summing those zeros to other zeros. SPVI exploits the same principle as all sparse linear algebra software libraries — that an operation that is guaranteed to produce a known result (zero), can be skipped altogether resulting in a performance improvement in time, memory use, and power consumption. By parallelizing computations using a GPU, and replacing linear algebra operations with GPU operations that are specifically optimized for sparse matrix-vector algebra, it is shown that a significant improvement in performance gain beyond the current state-of-the-art is achieved.

3.3.3 SPVI Algorithm

A pseudocode description of the SPVI algorithm is shown in Algorithm 1. The key features of SPVI that enable its high performance are: (1) a parallel deployment scheme that takes full advantage of as many GPU cores, blocks and threads as are available, (2) the consolidation of $(N_S N_A)$ independent operations into one large parallelized operation, and (3) the use of a single compressed sparse matrix in place of the N_A large STMs that are typically required.

3.3.3.1 State Transition

The computation of the product $\boldsymbol{M} \cdot \underline{V}^{n-1}$ in Equation 3.2 is efficiently implemented in SPVI using a sparse Matrix-Vector multiplication. The sparsity in the transition matrices is exploited by the conversion of the large and sparse $\boldsymbol{M}$ to a much smaller, densely packed $\boldsymbol{M}_s$ on lines 1 through 5. The sparse matrix $\boldsymbol{M}_s$ is created in the Compressed Sparse Row Matrix format. This conversion only needs to be performed once at initialization. Details on this sparse matrix format, as well as a thorough analysis of the history and performance advantages of performing

ALGORITHM 1: Sparse Parallel Value Iteration (SPVI).

```
Input: $\mathcal{S}$, $\mathcal{A}$, $R(s_i, a)$, $P(s_j|s_i, a)$, $\beta$, $\tau$
Output: $\underline{\pi}$
1  Compute $N_{NZ}$, the number of non-zero elements in $\boldsymbol{M}$
2  Allocate memory for $\boldsymbol{M}_s$, a sparse matrix sized for $N_{NZ}$
3  for each element in $\boldsymbol{M}$ do
4      if element is non-zero, add it to $\boldsymbol{M}_s$
5  end
6  $\underline{V}^0 \leftarrow \underline{0}$
7  $n \leftarrow 0$
8  repeat
9      $n \leftarrow n + 1$
10     $\underline{T} \leftarrow K\_SPARSE\_MULT(\boldsymbol{M}_s, \underline{V}^{n-1})$
11     $\underline{Q} \leftarrow K\_SAXPY(\beta, \underline{T}, \underline{R})$
12     $\underline{V}^n, \underline{\pi}^n \leftarrow K\_MAX\_REDUCE(\underline{Q})$
13     $\underline{N} \leftarrow K\_INF\_NORM(\underline{V}^n, \underline{V}^{n-1})$
14     $\Delta = MAX(\underline{N})$
15 until $\Delta < \tau$
16 $\underline{\pi} \leftarrow \underline{\pi}^n$
```

sparse Matrix-Vector multiplications in GPUs can be found in [43].

A sparse matrix format is a format for a data structure that represents a matrix. However, instead of the standard approach for matrix storage (a serialization of each element in the matrix regardless of its value), a sparse matrix structure contains an array of just the non-zero elements, along with two other arrays that indicate where those elements are located in the matrix. The format implicitly assumes that elements not specified are zero by default. In this form, a matrix can be represented with no loss of information and if the sparsity of a matrix is high then the sparse representation can be much smaller than the matrix stored in a standard (fully serialized) format. Correspondingly, multiplying a sparse matrix by a vector can be much faster and memory-efficient if the sparsity is high.

In SPVI, the multiplication of a sparse matrix by a vector is performed using a CUDA kernel and is denoted by *K_SPARSE_MULT* in Line 10. This kernel is a standard sparse matrix-vector multiplication as provided by the GPU libraries CUSP (2016) and NVIDIA's cuSPARSE (2015).

Next, the discount factor and rewards need to be applied. After computing the product $\boldsymbol{M}_s \cdot \underline{V}^{n-1}$, the remaining steps can be implemented by scaling the product by a scalar β and then adding it to the vector $\underline{R}$. This algebraic operation is commonly referred to as a Single-Precision A $\underline{X}$ plus $\underline{Y}$ (SAXPY), and is very efficiently implemented on most linear algebra packages. Several CUDA linear algebra packages (e.g., CUBLAS) provide a highly optimized parallel execution version of SAXPY, which is used in SPVI and referred to here as *K_SAXPY* in Line 11.

3.3.3.2 Action Selection

The N_S elements of the Value Function $\underline{V}^n$ and policy $\underline{\pi}^n$ are computed from $(N_S N_A)$ elements of the $\underline{Q}$ vector, which constitutes the selection of the optimal action for a given state. This computation is implemented with N_S parallel deployments of an N_A to 1 dimensionality reduction kernel invoked in Line 12 of Algorithm 1. The execution configuration for the kernel is set up to launch one kernel per MDP state. The kernel computes the maximum value (and the action associated with it) from a subset of $\underline{Q}$, striding across the vector only on the elements associated with the state assigned to the given kernel deployment.

3.3.3.3 Stopping Criteria

In order to evaluate the stopping criteria for SPVI, the infinity norm of the incremental approximations to the Value Function must be computed. This operation is represented by lines 13 and 14 of Algorithm 1. Line 13 is an execution of N_S parallel deployments of a reduction kernel. The kernel is deployed with an execution configuration of N_B CUDA blocks, where N_B is as large as possible for the specific CUDA hardware that is available.

As a result of this execution configuration, the reduction kernel of Line 13 provides a reduction from two length N_S vectors to one length N_B vector. Line 14 is a maximization loop run on the CPU. This use of a loop on the CPU does not have much impact on the execution time because the length of the vector $\underline{N}$ is N_B, which is typically much smaller than N_S.

3.4 Applications

The previous section detailed the methodology used by SPVI to exploit sparsity in MDPs. This section begins with a listing of the sparsity found in several MDPs encountered during this research, followed by discussion and justifications on why this is a typical finding for real-world MDPs.

3.4.1 Survey

This section contains a survey of several embedded systems challenges that are solved with MDPs. For these MDP solutions, the sparsity of the *STM*s were computed or estimated. The results are shown in Table 3.1. As is evident from the table, there is a high level of sparsity in each of these MDPs.

Problem	Domain	Sparsity
Coffee Robot	AI/Robotics	96.9%
Russell/Norvig Maze	Navigation/Planning	99.7%
ACAS	Avionics	99.1%
SPC	Sensor Networks	99.8%
MRCS	Cognitive Radio	90.9–93.3%

Table 3.1: Sparsity in surveyed MDPs.

In the Coffee Robot problem [15], researchers detailed a classic problem used in Artificial Intelligence and Robotics. The problem contains a decision framework used by a notional agent (the “robot”) that leaves an office building to buy coffee for its owner. In the Russel/Norvig Maze [35], researchers create a grid representing a floor plan, and show how MDPs can be used to solve indoor navigation problems. Since the original problem was an illustrative grid of size 3x4, the analysis here includes an extension of the grid to be much bigger in order to make the problem more realistic. In the Airborne Collision Avoidance System (ACAS) [46], MIT researchers developed an MDP for collision avoidance on commercial airliners.

The Solar Powered Computer (SPC) [2] problem is a challenging dynamic power management problem. The MDP-based Reconfigurable Channelizer System (MRCS) presented in Chapter 2 is a signal processing system that uses an MDP to dynamically reconfigure itself for a given environment and use case.

The examples listed above show anecdotally that sparsity can appear prominently in real-world MDPs. It is reasonable to conclude that this is a typical finding, not just a handful of outliers, in the context of MDP problems for embedded systems. Reasons supporting this conclusion are explored in Section 3.4.2 and Section 3.4.3, where common forms of sparsity that arise are described.

3.4.2 Multivariate State Spaces

In Section 2.4.4, the concept of a multivariate state space was defined. From MDP surveys in the literature it was observed not only that this type of state space is very common, but also that MDPs with this type of state space exhibit a high degree of sparsity. One reason that sparsity results from this is that an action rarely affects all of the state variables simultaneously. Rather, it may have an effect on one or only a few of the state variables, and thus the probability of a transition from a given state to all possible combinations of all of the other state variables is very unlikely or impossible. Rather, only a small subset of the combinations can be reached and this gives rise to sparsity in the *STM* for that action. This type of sparsity occurs very clearly in the Coffee Robot and MRCS problems.

For example, in the Coffee Robot problem, the state space consists of the combination of several Boolean variables of the form listed below. Each of these is either true or false at any point in the state space.

- O: The robot is located in the office.
- W: The robot is wet.
- U: The robot has an umbrella.
- R: It is raining.
- HCR: The robot has a coffee in its possession.
- HCO: The owner has a coffee in its possession.

The MDP actions for this problem consist of simple, direct actions. For example, one action is to pick up the umbrella if it is in the office. This action only affects the state variable U (umbrella); it does not change the robot's location, whether it is wet or not, etc. When the *STM* is constructed for this action, any states that correspond to a change in these other state variables is an unreachable state. This results in a large percentage of the *STM* being populated with zeros — representing that the probability of reaching those states is zero.

3.4.3 Sequential Decision Problems

Another reason to expect sparsity to be found in many real-world MDPs is that MDPs are often used to solve sequential decision problems. By definition, this implies having to make many small decisions to reach a desired long-term goal or state. For example, navigation problems usually involve a sequence of decision points where the agent must decide to turn left, turn right or go straight. These decisions move the agent from one location to another (nearby) location. The set of possible locations that can be reached by one decision makes up a tiny fraction of all of the possible locations in the entire state space. If there was an action available to simply jump to the destination state in one hop, then the navigation problem would not be very difficult to solve algorithmically (and thus an MDP would not be required).

However, this is usually not the case, and often a desirable long-term goal or state can only be reached through a sequence of decisions that create a trajectory through many small and incrementally overlapping subsets of immediately reachable states. When the STM is constructed for each action, only the tiny subset of states that are immediately reachable are populated with non-zero transition probabilities. The rest of the large matrix is filled with zeros. For this reason, it is reasonable to conclude that sparsity is an inherent property of MDPs when used to solve these types of problems. This type of sparsity occurs very clearly in the Russell/Norvig Maze, ACAS and SPC problems.

The following subsections contain discussions of specific case studies to show

how MDPs can be used to enable runtime adaptation in complex scenarios, and give more detailed illustrations of *STM* sparsity. These case studies were implemented in the MDP testbed and used to generate performance benchmarks that are summarized in Section 3.5.

3.4.4 Case Study: Solar Powered Computing

In recent years, there has been strong interest and optimism in the design of systems that can run purely off of a photovoltaic energy harvesting source (e.g., see [5, 54]). These energy sources are commonly combined with an energy buffer such as a supercapacitor or a rechargeable battery. A primary design goal for this class of systems is to design them in such a way that they can run indefinitely on the energy provided by the harvesting source alone, a property known as being *energy neutral* [55].

This goal is challenging because solar energy is time-varying and uncontrollable. Furthermore, there is no solar energy at night. In order to keep a system powered at times other than those of high incoming solar energy, it is necessary to use a suitable power management scheme that is well matched to the power requirements of the processing system, and also to the performance goals of the application. Additionally, an intelligent power management scheme might employ strategic conservation of energy stores to be used at a later time of the day.

An MDP was created to dynamically control the power of an embedded computing system closely mimicking the setup in [2]. The performance of this MDP-based controller was compared to the competing technique proposed in that work, which solves a constrained optimization problem using Linear Programming. The performance of the MDP was nearly identical to that of the Linear Programming approach. It is important to note that the authors in [2] cited the Linear Programming solver as too computationally intensive to run in their resource constrained embedded system, reinforcing the theme that often there are not enough computing resources to house intelligent decision frameworks in systems that can benefit from them.

3.4.4.1 State Space and Action Space

The state space for this problem is defined using the multivariate approach described above in Section 3.4.2. The state space for the system is represented as:

$$\begin{aligned} &s = (TD, SC), \\ &DOM(TD) = \{1, 2, \ldots, N_{TD}\} \\ &DOM(SC) = \{k/N_{SC} \mid k \in \{1, 2, \ldots, N_{SC}\}\} \end{aligned} , \tag{3.3}$$

where TD represents the discretized Time of Day (in hours), SC represents the discretized State of Charge (SC) of the energy buffer, and N_{TD} and N_{SC} are positive integers. The SC is the equivalent of a fuel gauge for the energy buffer, with the units being percentage points (0%=empty, 100%=full). Both TD and SC are inherently continuous valued quantities (time and energy) that are mapped into discrete sets through a coarse-grained quantization so that they can be incorporated into a discrete state MDP.

In order to manage power, the MDP action regulates the amount of application-specific processing that the system performs. This is assumed to be in the form of a duty-cycle. In other words, a percentage of time can be spent in a productive state versus a sleep or low-power state. The action $a \in \mathcal{A}$ is the choice of the duty cycle, which is discretized as shown in Equation 3.4:

$$\mathcal{A} = \{k/N_A\} \mid k \in \{1, 2, \ldots, N_A\}. \tag{3.4}$$

3.4.4.2 State Transition Matrix

The *STM*s are defined as shown in Equation 3.5. This *STM* formulation utilizes the fact that the TD variable is independent of both the SC variable and the action. This allows for *factorization* (as described in Section 2.4.4) to be employed in the state space, a concept that plays a crucial role in the compact modeling and efficient solving of MDPs.

$$\begin{aligned} P(s_j|s_i, a) &= P(TD_j, SC_j|TD_i, SC_i, a) \\ &= P(TD_j|TD_i)P(SC_j|TD_i, SC_i, a) \end{aligned} . \tag{3.5}$$

The entries for the transition probabilities of the TD variable are then computed as in Equation 3.6. Here, a simplification is made due to the discretization width of the TD variable being equal to the control frame interval. Thus, for each iteration of the control frame, the time of day is guaranteed (with probability 1) to increment by 1 hour.

$$p(TD_j|TD_i) = \begin{cases} 1, & TD_j = mod(TD_i, N_{TD}) + 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

To compute the entries for the probabilistic transition of the SC variable, knowledge of how the SC will be affected by each control action is required. For this, the historical platform power consumption data is used, as well as the predicted incoming solar energy, as shown in Figure 3.1. The Platform Power Model block in this figure refers to a running estimate of how much average power will be consumed by each processing configuration (i.e., each MDP action), as predicted from the logged data. The mathematical details of the SC transitions are omitted here for conciseness.

An illustration of the underlying state transition structure that results in this MDP formulation is shown in Figure 3.2. Each circle represents a state in the state space, with the shaded circles denoting one possible trajectory. The arrows represent the reachable states (i.e., the transitions that have non-zero probability of occurrence). It can be seen from this formulation how sparsity is a central characteristic of the STM.

A non-sparse STM would be a scenario where most or all of the circles in the figure can be reached from any given circle. Such a scenario is not possible for two reasons. First, for any circle (starting state) only the subset of circles consisting of the next time frame (the column immediately to the right) are reachable. Thus, from the TD component of the state space alone, the sparsity of the STM will be at least $1-(1/N_{TD})$. Second, only a few rows of the next column are reachable, not

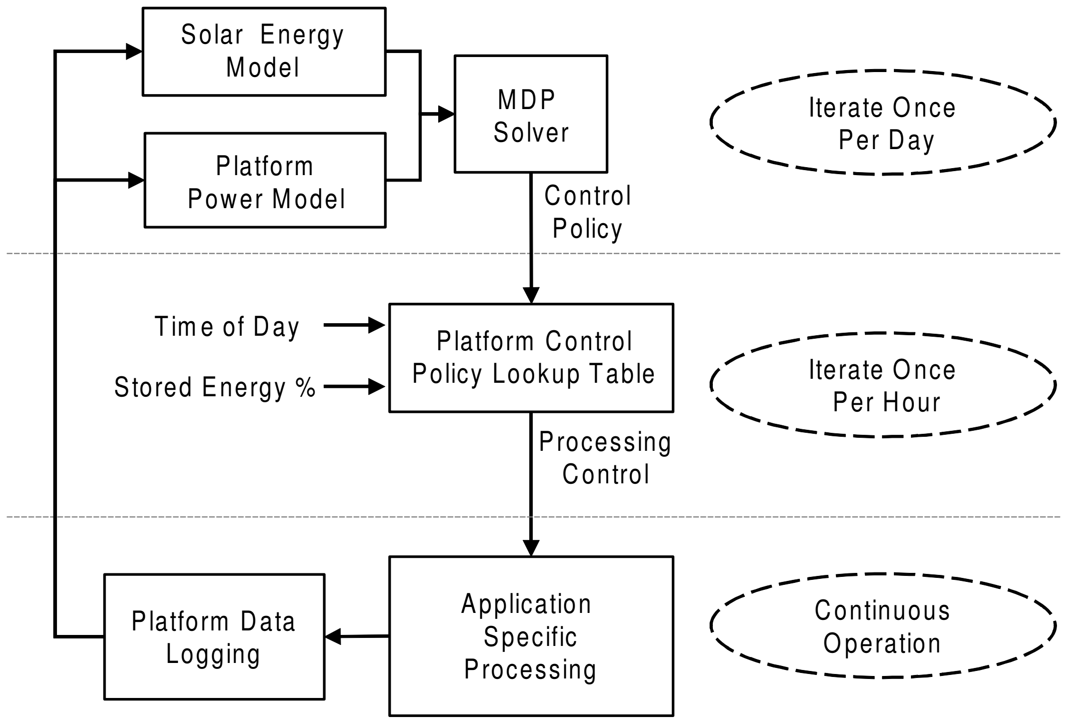

Figure 3.1: Block diagram of MDP-based dynamic controller for a solar-powered computing system.

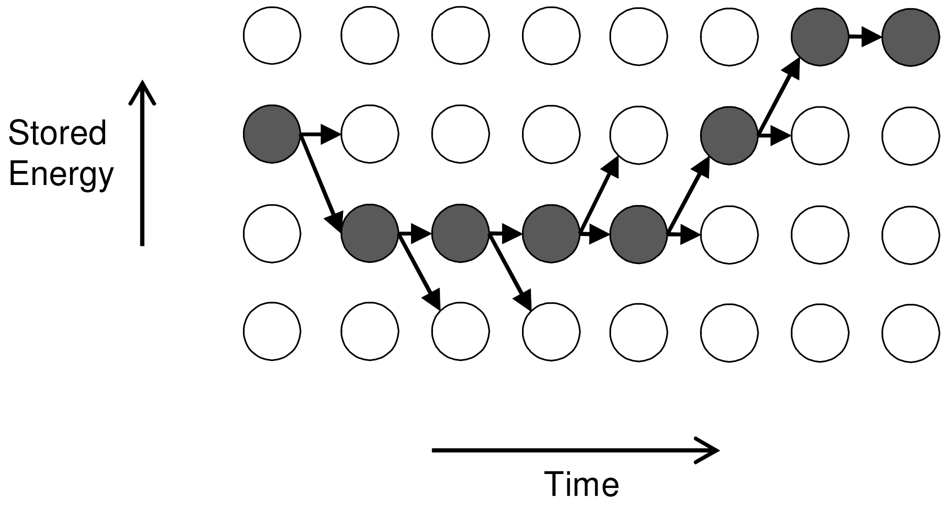

Figure 3.2: Time-Energy state grid and transitions.

the entire column. This further increases the sparsity.

In these experiments, it was found that sufficient granularity in the discretization of the state space (in order to attain good decision making performance) could be reached with $N_{TD} = 24$, and $N_{SC} \geq 50$. With the state spaces sized in this way, the total *STM* sparsity is at least 99.8%, as is listed in Table 3.1.

3.4.5 Case Study: MDP-Based Reconfigurable Channelizer System (MRCS)

Chapter 2 detailed significant advantages to using an MDP to derive dynamic reconfiguration policies for use on channelizers in stochastic environments. The MDP-based approach for digital channelizer design optimization resulted in increased robustness when used to periodically re-optimize the system policy specifically for the external environment it is being used in. However, in that chapter the proposed method utilized a single MDP-based policy that was generated by invoking an MDP solver once offline (in MATLAB) and not in the target system. In order to predict the effectiveness of SPVI to address this limitation, the MRCS is revisited here to analyze its sparsity.

The state space for this problem was defined using the multivariate approach described in Section 3.4.2. The state variables consist of the channelization requests, combined with the available processing configurations for a given processing platform. The state space for the system is defined in Equation 2.13.

In this system, the state space contains all possible combinations of channelization requests with all possible platform configurations. The action space is defined by all combinations of the processing configuration variables, such that the MDP can select which processing configuration to use at any given time. Thus, the MDP actions are the set of selectable configurations and $N_A = |DOM(\underline{CF})|$.

The channelization requests are exogenous and not under the control of the MDP. However, the processing configuration is fully under the control of the MDP. This scenario allows the STM to be defined as was shown in Equation 2.14. In this formulation, the $\underline{CR}$ state variables form their own independent uncontrollable Markov chain embedded within the larger MDP state space and transition matrices.

A notional diagram of the state transition diagram for an arbitrarily selected action is shown in Figure 3.3, for a scenario where N_A is 3. This diagram shows that when Action 3 is selected by the MDP policy, the system transitions with a probability of 1 to the region of the state space corresponding to platform configuration $\underline{CF}$ 3. Anytime Action 3 is selected by the policy, the system can transition within different values of the $\underline{CR}$ variables as prescribed by the factored transition probabilities $P(\underline{CR}_j \,|\, \underline{CR}_i)$, however it cannot transition to any region of the state space corresponding to a different platform configuration.

As a result, the transition probabilities to all other states that do not correspond to configuration $\underline{CF}$ 3 are zero, and for this reason sparsity is an inherent part of this MDP formulation. Since only one selected configuration is reachable for a given action, an STM sparsity of at least $1 - (1/N_A)$ results. The system in Chapter 2 uses N_A values ranging from 11 to 15, and thus a sparsity ranging from 90.9% to 93.3% results.

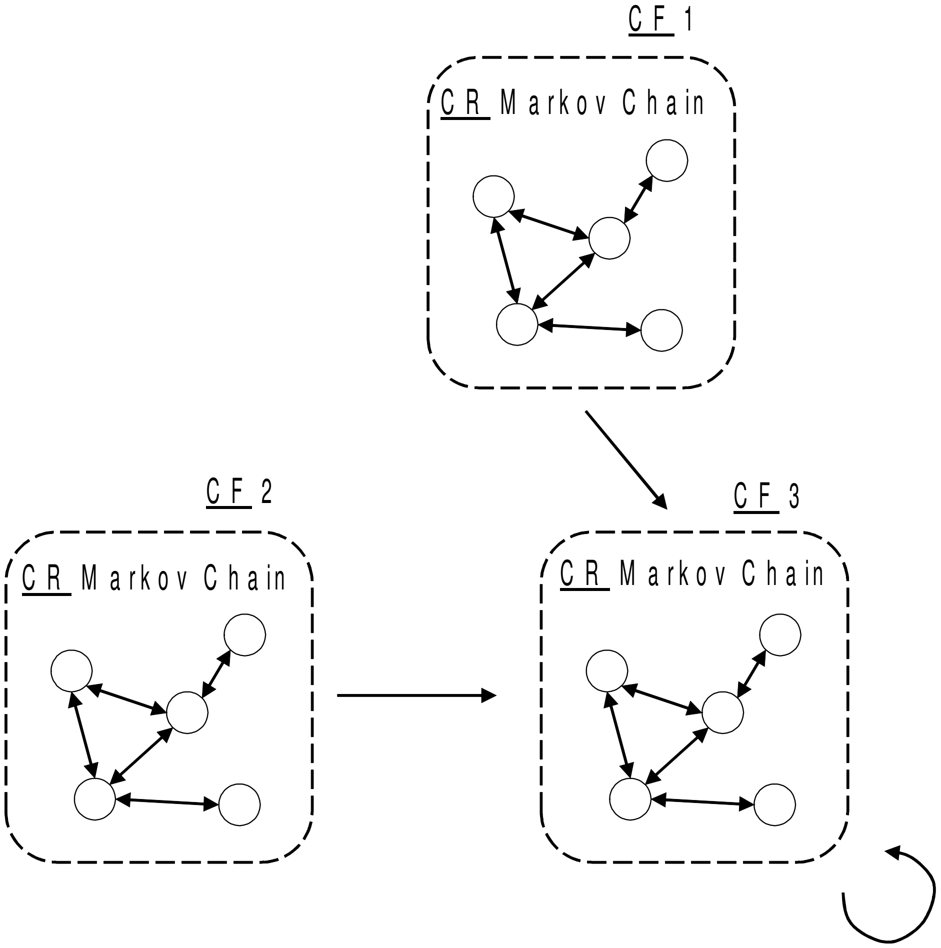

Figure 3.3: Channelizer state transition diagram for Action 3.

3.5 Results

3.5.1 Experimental Setup

In order to objectively measure the performance of SPVI compared to alternative methods, a testbench was created using the NVIDIA Jetson TK1 development board. This board consists of an NVIDIA Tegra K1 SoC, which is a size, weight and power optimized processor used in many embedded and mobile applications. The SoC contains a Quad-Core ARM Cortex A15 CPU running an embedded Linux distribution. The setup allows CPU-only programs to be run, which is an important part of the comparative benchmarking, which compares CPU-only with GPU-accelerated approaches. The GPU acceleration is made possible on-demand by enabling the K1 SoC's on-chip Kepler GPU, which contains 192 CUDA cores and can be programmed and controlled directly from a user space application on the embedded Linux distribution.

The testbench contains implementations of three of the MDPs introduced in Section 3.4. The MDPs are used as the input to multiple MDP solver implementations. In general, it was found that the competing solver algorithms all produced roughly the same policy output, and thus the comparisons are purely in terms of the computational resources expended in finding the policies.

Through the use of this testbench, it was possible to keep all test details consistent and change only the solver algorithm. This allows for a fair comparison of solvers using consistent and objective experiments. The solvers selected for comparison are described in Section 3.5.2.

3.5.2 Algorithm Comparison

Three algorithms were used for benchmarking, which are referred to here as VI, Thrust-VI and SPVI. The first algorithm is a CPU-only implementation of Value Iteration, which is referred to below as VI. This was created directly from Equation 3.1. This implementation represents a straightforward single-threaded implementation of the classical Value Iteration algorithm and was used purely as a baseline for comparison with the other algorithms.

Next, a survey of the literature was performed to find the current state-of-the-art in high performance MDP solvers. Several algorithms and open source packages were analyzed. The search was restricted to open source candidates that were available in C or C++ source code, and avoided those in Java, MATLAB or Python. This decision was made so that the results could be applied with no caveats to resource constrained embedded systems that are not able to support the runtime requirements of higher level languages.

The solvers that were obtained or implemented are: SPUDD [47], AAPL [56], pomdp-solve [57], and Thrust-VI [42]. In the case of SPUDD, AAPL, and pomdp-solve, open source software was available to download and use in the experiments. For Thrust-VI, no software was available but the algorithm description in the literature was detailed enough that direct implementation from pseudocode was possible without any ambiguity. After some experiments and analysis, it was determined that Thrust-VI had the best performance among candidates. Thus, Thrust-VI was identified as the current state-of-the-art and chosen for comparison against the SPVI algorithm.

Many details and logistical matters involved in searching for open source MDP solvers were identified as important issues during this work. Discussion of these issues, the criteria used to determine the state-of-the-art, and a more in-depth analysis of the solver survey is presented in Chapter 5.

3.5.2.1 Thrust-VI

Thrust-VI is a GPU accelerated version of Value Iteration. The algorithm uses the open-source NVIDIA Thrust library. Thrust is a C++ parallel algorithms library that allows programmers to write portable C++ code, and then execute that code on CUDA GPUs, OpenMP systems, and other parallel execution platforms. Thrust-VI is an implementation of the classic Value Iteration algorithm, using Thrust's parallel functions. The finding that this algorithm outperforms other more algorithmically complex alternatives is an interesting outcome. It suggests that decades of novel and clever algorithmic advances in this field were surpassed by simply exploiting data parallelism with GPUs — a completely different approach with fewer restrictions.

Both SPVI and Thrust-VI leverage GPUs for execution time acceleration. The most significant difference between these two is that SPVI uses sparse matrix representations and sparse matrix-vector operations and Thrust-VI does not.

3.5.3 Measurements

In the experiments, the solver execution time was measured using Linux's native timing support. The results are listed in Table 3.2. Both GPU-accelerated solvers (Thrust-VI and SPVI) were considerably faster than the CPU-only Value Iteration solver.

MDP	VI	Thrust-VI	SPVI
Solar Powered Computer	28.48	6.62	1.60
Russell/Norvig Maze (30x100)	31.71	7.33	1.62
MRCS Channelizer	44.66	9.14	3.19

Table 3.2: Solver execution time (seconds).

SPVI significantly outperformed Thrust-VI in all of the MDPs that were tested. The execution time reduction from Thrust-VI to SPVI was 76% for the Solar Powered Computer, 78% for the Russell/Norvig maze (with a grid size of 30 rows by 100 columns), and 65% for the MRCS Channelizer. Although Thrust-VI did utilize the 192 CUDA cores effectively in all of the MDPs, it spent a lot of time multiplying elements by zero for the reasons detailed in Section 3.4. SPVI directly addresses this inefficiency via the use of sparse linear algebra optimizations and as a result is able to reduce execution time considerably.

3.6 Conclusion

In this chapter, the utility of sparse data structures were demonstrated for solving a large class of MDPs that are relevant to embedded computing systems. Reasoning was presented conceptually on why sparse matrices arise in MDP formulations, and significant performance gains were presented from exploiting this sparsity in solver implementations on a mobile Graphics Processing Unit (GPU). The ability to solve MDPs efficiently on resource- and power-limited mobile GPUs enables novel applications in which MDP solvers are embedded in the applications rather than being restricted to offline use.

Additionally, the proposed methods reduce restrictions and a priori assumptions that are required by compact MDP solvers that have been developed previously by researchers. This advancement can lead to a wider consideration of MDPs in embedded computing systems where they previously may not have been feasible or practical.

Useful directions for future work include further acceleration by decomposing MDP transition matrices up into groups of smaller matrices that can be multiplied utilizing faster GPU block shared memory (instead of slower global memory). Another direction is to investigate sparsity-driven acceleration of the other two classical solver algorithms: Policy Iteration and Modified Policy Iteration.

Chapter 4

Runtime Adaptation in Wireless Sensor Nodes Using Structured Learning

4.1 Introduction

This chapter presents a novel algorithm and a detailed application of techniques that enable the fast and efficient use of MDPs in real-time on resource constrained Cyber Physical Systems (CPSs). In this context, a resource constrained CPS is defined as a distributed system containing components that have embedded computers whose physical resources are constrained to be significantly below what is available in a typical, consumer-grade desktop or laptop computing system. These limits are typically imposed in order to reduce the size, weight and power, as well as cost (SWAP-C) of each unit. While this is a relative definition of a resource constrained CPS, in terms of present technology it can be defined as an embedded system that would typically have less than 1MB of RAM, less than 10MB of non-volatile storage, and a single-core microcontroller with a clock speed under 100MHz.

In previous chapters, MDPs have been explored as a means to control computing systems at runtime in ways that are more dynamic, robust and adaptable than alternatives. Using MDPs, engineers can create systems that effectively learn and reason using models of their own system dynamics, observations of their own inherent limitations and effectiveness of their actions towards reaching application-level goals. In this context, these systems exhibit a level of *self-awareness* in their behavior, with the ultimate design goals being continual autonomous optimization that leads to higher levels of runtime resiliency, robustness and efficiency.

When seeking to develop self-aware systems, researchers have recently turned to Reinforcement Learning (RL) [58], a field of Machine Learning that uses MDPs in restricted ways. However, this chapter explores an important class of CPSs that are not well served by current RL techniques. Specifically, this class of systems is one where some components of the system's dynamics are known at design time, and the rest are unknown at design time or expected to be time-varying at runtime.

In general, RL frameworks do not try to learn the effect that control outputs have on the system's state. Instead, they seek to use runtime observations to find a relationship between control outputs and rewards. In general, however, a critical part of this relationship is how the control output affects the state of the system being modeled. In RL frameworks, this causality is implicit in the modeling abstraction and not defined nor learned explicitly. This method works well in many cases, for example in large systems where the state dynamics are too large and complex to be considered or modeled explicitly. However, this chapter shows that departing from this conventional method can be useful for resource constrained CPSs that have more manageable state spaces. In particular, if engineers possess a priori knowledge about how some of the control outputs might affect the system state, it can be advantageous to codify that knowledge into the learning algorithms at design time. The approach proposed in this work provides models and algorithms that enable designers to exploit a priori knowledge in this way.

Motivated by this deficiency, an alternative class of MDP-based system modeling techniques are defined, which are referred to as *Compact MDP Models* (*CMMs*), and CMM-based approaches are developed as an alternative to RL for design and implementation of adaptive CPSs. In general, it is envisioned that certain CPSs are better suited for RL, while others are better suited for CMM. Through this work, the causes of why one of these two approaches might be better over another for a given application are explored, with the goal of improving understanding to allow designers to pick the better option.

Building on the work in the previous chapters, this chapter contributes the following additional content:

- A detailed survey of techniques from the literature that enable the compact use of MDPs on resource constrained systems. This survey leads to the definition of a class of CMM methods which encapsulates several different approaches that are useful in streamlining the application of MDPs to CPS systems.

- The introduction of the Sparse Value Iteration (SVI) algorithm — a variation of the SPVI algorithm presented in Chapter 3. While SPVI is a parallel processing algorithm developed for GPUs, in this chapter a scaled down variation is presented that runs effectively on small, single-threaded Microcontrollers (MCUs).

- A detailed example of how to apply MDP-based techniques to design a wireless sensor CPS.

- The results of a performance simulation, illustrating the differences between a CMM-based design approach compared to Q-Learning, a popular alternative technique from the Reinforcement Learning (RL) literature.

- An empirical study of an MDP solver running on a resource constrained MCU, including measurements of data storage requirements, execution time and power consumption.

- A power consumption model for an LTE-M wireless modem, derived from experimental lab measurements taken on a live wireless Internet Protocol (IP) data network.

The remainder of the chapter is organized as follows. A cursory review of the history of techniques for controlling CPSs is presented in Section 4.2. Section 4.3 contains a survey of recent advancements in CMMs. Section 4.4 presents Structured Learning with Sparse Value Iteration, a novel method for CMM-based design. In Section 4.5, a case study of a wireless sensor CPS is detailed, to explore the challenges and trade-offs inherent in creating an efficient control policy. Section 4.6 contains an illustration of how CMMs can be used to solve the design problem introduced in Section 4.5 along with a comparison of that approach to a competing RL-based approach. Finally, Section 4.7 contains simulations of the runtime performance and Section 4.8 contains the results of an embedded system implementation of the competing techniques. Section 4.9 concludes the chapter with a discussion of the results and directions for future work.

4.2 Background and Related Work

Chapter 1 and Section 3.2.1 contained a discussion of the challenges involved in applying MDPs to control computing systems at runtime. To overcome the limitations of MDPs with these goals in mind, two main approaches have been pursued: RL and CMMs. RL essentially tries to arrive at the policy without explicitly modeling all of the MDP components or invoking a solver. On the other hand, CMMs are approaches that do define all of the MDP components and invoke a solver, but do so via algorithmic optimizations that significantly reduce computational requirements. These two alternatives are sometimes referred to as model-free and model-based RL, respectively. Henceforth in this chapter, the abbreviation "RL" refers to model-free RL unless otherwise stated.

These two categories — RL and CMM — of techniques are described in Section 4.2.1 and Section 4.3, respectively.

4.2.1 Reinforcement Learning

Reinforcement Learning (RL) [58] is an area of machine learning that enables systems to formulate optimal decision policies using observations of rewards that are received for previous decisions at runtime. These techniques use MDPs as the framework for formulating the decision problem, but seek to learn the optimal policy directly using observations of rewards in response to decisions, rather than through the explicit definition of all of the MDP components followed by invocation of an MDP solver.

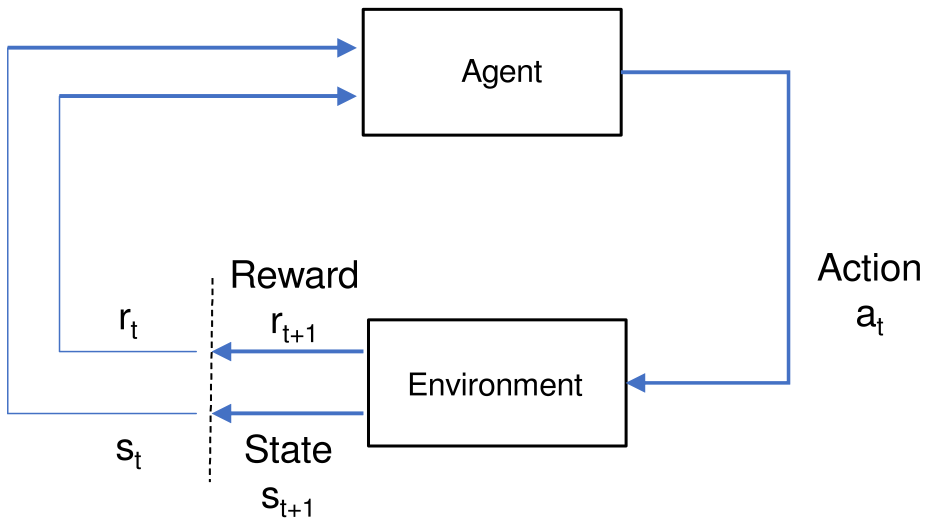

Figure 4.1: Block diagram of reinforcement learning paradigm.

More specifically, an RL framework typically contains the top level block diagram shown in Figure 4.1. The learning takes place by some agent, which is responsible for selecting an action out of a set of actions, given a system state. This selection is typically done in a discrete-time setting and iterated at a fixed rate. Each selected action (in a given state) leads to some consequence in the environment, and that causes it to transition into a new state in the state space. The selected action and transition to that new state are associated with a scalar reward, which is fed back to the agent (positively or negatively). The agent in turn considers the reward it has been given along with the new state that resulted, and again selects the next action, repeating indefinitely.

As mentioned above, these techniques are sometimes referred to as model-free learning. Model-free learning techniques possess the advantage of completely bypassing the need to maintain large STMs, and run computationally intensive

MDP solvers.

However, this advantage comes at a cost, as the consequences of all actions taken in all states have to be learned and periodically updated, even those that are constant and known a priori at design time. This learning comes at the cost of occasionally having to make random decisions at runtime to explore the effect of alternative decisions [59]. This cost is a central drawback of model-free techniques, and is associated with a complex trade-off called the *exploration versus exploitation trade-off* [58].

One popular RL technique that has shown promising results is Q-Learning [44, 60, 61, 62]. In Q-Learning, a scalar value “Q” is assigned to each action in each state, and referred to as the Q function. This function represents the average future rewards that can be expected by taking a given action in a given state. The Q-Learning method continually learns and updates this Q function using a simple technique called the method of Temporal Differences (TD) [63], and uses it to formulate an optimal control policy that is based entirely on the system and environment involved. As the environment and the system's dynamics change at runtime, the policy changes with it.

In one example [44], an Adaptive Power Management (APM) hardware module using Q-Learning was used to put a microcontroller in and out of low power states, and resulted in a learning controller that managed power transitions better than an expert user. In another example [60], Q-Learning was used to optimize the throughput of an energy harvesting wireless sensor node while meeting challenging constraints.

4.2.1.1 Stability

Under the control scheme proposed in this chapter, an MDP-based control policy that varies at runtime is used to control a computing system. This scenario requires a more involved analysis of stability compared to the scenario discussed in Section 2.4.5, where the MDP-based control policy was fixed at runtime. Both the proposed method, and the competing Q-Learning method are subject to the same questions of whether the resulting time varying MDP-based control has the potential for exhibiting any runtime instability. These questions fall under the category of stability analysis of time varying MDP-based control, which is an active area of ongoing research in the RL community. The earliest known work on this topic is [64], where the concept of Lyapunov functions are applied to guarantee stability in time varying MDP-based control. Lyapunov functions are a fundamental tool used in control systems theory to analyze the stability of systems evolving through time. These concepts are outside the scope of this thesis, and present an interesting area for future work. The reader is directed to [65] and references therein for the latest research in this area.

In the next section, an overview of CMM methods is presented, which can be viewed as *model-based* methods that are designed with an emphasis on streamlining computational efficiency. Recent advancements in this area are surveyed that result in performance on par with Q-Learning.

4.3 Survey of Compact MDP Models

As mentioned previously, the deployment of MDPs in resource constrained systems has typically been limited to usage modes where the MDP modeling and solving are done offline, and only the resulting policy is stored in the runtime system. The goal of moving the MDP model and solver into the runtime system by mitigating the longstanding barriers has been a common goal among many researchers over the years, resulting recently in very creative and effective techniques. In this chapter, this category of approaches is referred to as compact MDP models (CMMs), given that they provide a smaller or computationally optimized representation of the system in question than compared to a direct implementation of the MDP's data structures. The effectiveness of these recent developments, especially when applied in combination with one another, leads to questioning the conventional approach of limiting consideration to model-free RL approaches in the implementation of resource constrained, MDP-based systems.

Some of these CMM techniques seek to reduce the storage size of the MDP's data structures by exploiting some structural component embedded within the MDP (e.g., see [15, 38, 47, 48, 49]). Other techniques involve modeling approaches that reduce the MDP state space via generalization and abstraction of system dynamics (e.g., the approach in Chapter 2). Another approach has been to keep algorithms and data structures as is, and take advantage of recent advancements in parallel processing using embedded GPUs, for example [42] and the algorithm presented in Chapter 3.

In this chapter's case study, three CMM techniques are utilized. Each of these has been introduced in a previous section: transition states (Section 2.4.3), factorization (Section 2.4.4), and exploitation of sparsity (Section 3.2.2). The significant advantages each of these provides becomes evident when compared to both a direct classical MDP implementation and a competing model-free method.

4.4 Method

4.4.1 Structured Learning

Creating an MDP model on a computing system consists of defining the states and actions, the *STM*s, and the reward function for the given decision problem and its environment. The *STM*s are N_A stochastic matrices, each of size N_S by N_S (one matrix for each action). Each *STM* defines the probability of transitioning from the current state to any one of the possible other states, given an action. This is generally written as a discrete conditional probability distribution as in Equation 4.1:

$$p(s^{(n+1)}|s^{(n)}, a^{(n)}), \forall s \in \mathcal{S}, a \in \mathcal{A}, \tag{4.1}$$

which gives the probability of the system transitioning to state $s^{(n+1)}$ at time index $n+1$ given that it was in state $s^{(n)}$ and action $a^{(n)}$ was selected at time index n. The process of instantiating the *STM*s is the allocation of storage for $N_S^2 N_A$ numerical quantities and assigning them values from 0 to 1. Given this viewpoint, the methods in the literature can be grouped into two categories: the model-based approach where all $N_S^2 N_A$ terms are defined a priori and treated as constants, and the model-free approach, where none of the $N_S^2 N_A$ are defined and in fact storage for them is never even allocated.

In this chapter, these two are viewed as extremes of a continuum that has many other options. A blend between the two is proposed, where some of the *STM* terms are assumed to be known a priori, and others are not. More specifically:

$$p(s^{(n+1)}|s^{(n)}, a^{(n)}) \in \{\Gamma \cup \hat{\Theta}\},$$
$$\forall s \in \mathcal{S}, a \in \mathcal{A}, \tag{4.2}$$

where the implication is made that all of the probability values in the STMs come from one of two parameter subsets Γ and Θ. The set Γ is the set of STM entries that are fully known a priori and can be set to a fixed value at design time. The set $\hat{\Theta}$ contains the remaining matrix elements; they are either not known a priori or are expected to be time-varying at runtime. $\hat{\theta} \in \hat{\Theta}$ is used to denote the latest value of a running of estimate of the true value, for each parameter $\theta \in \Theta$.

Rather than taking each entire STM as either constant, or completely unknown, a flexible middle ground is adopted and it is assumed to be partially known and partially unknown. In this way, the system model has some parts of it that are fixed, and other parts that are assumed to change over time. Then, the runtime adaptation process consists of learning only the set of parameters $\hat{\Theta}$, rather than the entire STMs. In this way, the model contains a mix of some predetermined structure from Γ) and some runtime learning (from $\hat{\Theta}$). This approach is referred to as Structured Learning in this chapter.

Structured Learning allows the designer to restrict how much effort is spent trying to learn unknown parameters, and results in higher overall awareness and adaptation performance for a certain class of CPS devices, as will be demonstrated in the case study. The advantage comes from being able to direct the system's learning efforts to be focused on the relevant parts of the problem, and prevent redundant attempts to constantly question and update assumptions about the system that a system designer knows will never change.

4.4.2 Temporal Difference Equations

The Structured Learning method defined above relies on a continual runtime learning of parameters. For this, a very simple technique is used, which is prevalent in RL: the use of weighted averaging through Temporal Difference (TD) equations, with the central concept shown in Equation 4.3:

$$\hat{\theta}^{(n+1)} = \hat{\theta}^{(n)} \cdot (1 - \alpha) + \theta^{(n)} \cdot \alpha, \tag{4.3}$$

where $\theta^{(n)}$ is an observed value of one of the parameters $\theta \in \Theta$ at timestep n, $\hat{\theta}^{(n)}$ is the value of the running estimate of θ at timestep n, and α is a learning rate parameter which controls how sensitive the running estimates are to individual observations. The method essentially consists of performing a low-pass filtering or smoothing operation on observed values, and thus maintaining a running estimate that tracks the latest observed values for a given parameter.

As the observations change over time, the running estimates track the changes while also reducing the effect of statistical outliers. Using TD, the Structured Learning method can ingest the latest observations of each of the parameters $\theta \in \Theta$, compute the running set of estimates for each $\hat{\theta} \in \hat{\Theta}$, and combine them with the constant set Γ to assemble the fully populated, partially time-varying STMs at any timestep.

4.4.3 Sparse Value Iteration (SVI)

In Structured Learning, the full set of STMs are instantiated. In order to obtain a control policy from this, an MDP solver must be invoked. Although Structured Learning is fully compatible with the SPVI algorithm that was presented in Section 3.3, it is not used in this chapter. This is because SPVI is a GPU algorithm, and the case study in this chapter is on a resource constrained system where it is assumed that a GPU is not available.

For this reason, a modified version of SPVI is introduced here. SPVI achieves runtime acceleration not only from the parallel processing performed on a GPU, but also from the use of sparse matrix representations and sparse linear algebra operations. Through experimental benchmarking, it was found that the sparse linear algebra techniques alone add performance benefits on single-threaded CPUs without the use of parallel processing on GPUs. The experimental results in the following sections show this to be true even on a resource constrained single threaded MCU, not just on the much more powerful processing environments of larger CPUs.

The acceleration is not as significant as when a GPU is available, however it is considerable enough to be a valuable technique for use on a wide range of CPU-only systems. The result of these findings is a new algorithm presented here, called Sparse Value Iteration (SVI). This algorithm is similar to SPVI, with the difference that SVI is suitable to run on single-threaded CPUs that are not equipped with GPUs.

The performance benefits of SVI over VI materialize as decreases in runtime, processing energy consumption, and memory footprint. The specific decreases in the experimental setup are detailed in Section 4.8. A pseudocode description of the SVI algorithm is shown in Algorithm 2.

ALGORITHM 2: Sparse Value Iteration (SVI).

```
Input: $\mathcal{S}$, $\mathcal{A}$, $R(s_i, a)$, $P(s_j|s_i, a)$, $\beta$, $\tau$
Output: $\underline{\pi}$
1  Compute $K_{NZ}$, the number of non-zero elements in $\boldsymbol{M}$
2  Allocate memory for $\boldsymbol{M}_s$, a sparse matrix sized for $K_{NZ}$
3  for each element $z$ in $\boldsymbol{M}$ do
4      if $z \neq 0$ add it to $\boldsymbol{M}_s$
5  end
6  $\underline{V}^0 \leftarrow \underline{0}$
7  $n \leftarrow 0$
8  repeat
9      $n \leftarrow n + 1$
10     $\underline{T} \leftarrow SPARSE\_MULT(\boldsymbol{M}_s, \underline{V}^{n-1})$
11     $\underline{Q} \leftarrow SAXPY(\beta, \underline{T}, \underline{R})$
12     $\underline{V}^n, \underline{\pi}^n \leftarrow MAX\_REDUCE(\underline{Q})$
13     $\Delta \leftarrow INF\_NORM(\underline{V}^n, \underline{V}^{n-1})$
14 until $\Delta < \tau$
15 $\underline{\pi} \leftarrow \underline{\pi}^n$
```

4.4.3.1 State Transition

The computation of the product $\boldsymbol{M} \cdot \underline{V}^{n-1}$ in Equation 3.2 is efficiently implemented in SVI using a sparse Matrix-Vector multiplication. The sparsity in the transition matrices is exploited by the conversion of the large and sparse $\boldsymbol{M}$ to a much smaller, densely packed $\boldsymbol{M_s}$ on lines 1 through 5. Then, a multiplication of the sparse matrix by a vector is performed using a subroutine denoted by *SPARSE_MULT* in Line 10. This subroutine is a standard sparse matrix-vector multiplication.

The sparse matrix $\boldsymbol{M_s}$ is created in the Compressed Sparse Row Matrix format, as was the case in SPVI. This conversion only needs to be performed once at initialization.

Next, the discount factor and rewards need to be applied. After computing the product $\boldsymbol{M_s} \cdot \underline{V}^{n-1}$, the remaining steps can be implemented by scaling the product by a scalar β and then adding it to the vector $\underline{R}$. This is the same SAXPY operation that was described in SPVI, and is also very efficiently implemented in CPU-based linear algebra packages. This subroutine is denoted here as $SAXPY$ in Line 11 of Algorithm 2.

4.4.3.2 Action Selection

The N_S elements of the Value Function $\underline{V}^n$ and policy $\underline{\pi}^n$ are computed from $(N_S N_A)$ elements of the $\underline{Q}$ vector, which constitutes the selection of an optimal action for a given state. This computation is invoked in Line 12 of Algorithm 2.

The subroutine computes the maximum value (and the action associated with it) from a subset of $\underline{Q}$, striding across the vector only on the elements associated with each state.

4.4.3.3 Stopping Criteria

In order to evaluate the stopping criteria for SVI, the infinity norm of the incremental approximations to the Value Function must be computed. This operation is represented by Lines 13 and 14 of Algorithm 2.

4.5 Application

In this section, a specific type of CPS is detailed, which forms the basis for the case study in the remainder of the chapter. The CPS is an embedded system with constraints on its size, weight and power (SWaP), containing the physical components shown in the following list.

- A sensor and/or actuator to interact with the physical environment.
- A wireless modem used to provide Internet access to the system.
- A low-power Microcontroller Unit (MCU) executing a program that controls the sensor and/or actuator as well as the wireless modem.
- An energy source that is used to power the system. The source can be a battery that needs to be replaced periodically, or an energy harvesting source (such as a solar panel paired with a rechargeable battery).

This type of CPS is expected to exist as one instance of a plurality of identical nodes in an installed base, and the nodes are connected to an application server via an Internet connection. The design aims to empower each node with the ability to optimize its performance for its own specific environmental conditions, rather than centralizing all of this optimization responsibility in the cloud. This approach is advantageous in terms of scalability, reliability and robustness.

The MCU runs an application-specific program that utilizes the sensor and/or actuator to interact with its physical environment. The application is typically multi-modal, in the sense that it does not always do exactly the same application-level operations at all times. The application may, for example, have two modes such as 1) the normal sensor and actuator operating mode, and 2) a firmware update mode where security patches and product updates are routinely downloaded to the CPS node. It is expected that the application could have more than two modes.

The CPS node's power source is typically very limited in its capacity. In the case of a solar panel, this may be due to a desire to keep the solar panel cost low and the size small. In the case of a battery, this may be due to a desire to maximize battery life in order to reduce how often the battery needs to be recharged or replaced. Regardless of the power source, the CPS node's application is generally tasked with carrying out its functions in the most energy efficient manner possible due to limitations in its energy source.

In order to maximize battery life, a low-power MCU is used, rather than a general purpose computing system. As a result of this, the CPS node will be limited in its CPU frequency, RAM size, and non-volatile storage capacity. This fact becomes important when considering the use of computationally intensive control algorithms. A more involved program requires more computing resources, which in turn requires the use of a more capable computing system that consumes more power (even while idle). Thus, a thorough consideration of control algorithms must analyze not just the control performance, but also the computational requirements that are needed to deploy it on a self-contained and resource-limited MCU.

The MCU is also tasked with controlling the wireless modem to enable communication typically with another Internet-connected device such as another CPS node or a cloud-based application server. In this case study, the focus is on one specific form of wireless network that is very common at the time of this writing: LTE-M (LTE for Machines), also known as LTE Cat-M1 [66]. This wireless protocol is a subset of the full LTE protocol, the wireless network that makes up the majority of cellular Internet connections at the time of this writing. LTE-M is a reduced form of the full protocol, and is designed specifically for resource constrained devices. In the following section, the power profile of an LTE-M modem is described in detail in order to illustrate the power management considerations an MCU controller must balance.

4.5.1 LTE-M

The LTE-M modem is usually the largest consumer of power in the processing and communication subsystems. When actively communicating with a cell tower, an LTE-M modem's average power consumption can be as much as 650 milliWatts. This quantity is very high relative to the consumption of other components in the CPS node. As a result, leaving the LTE-M modem powered on and connected to a cell tower continuously is usually not a feasible option for CPS nodes, from the point of view of maximizing the lifetime of a limited energy supply. As a result, the MCU must turn the LTE-M modem on and off strategically in order to stay within a limited energy budget.

In order to fully understand this power management challenge, a Sequans Monarch VZM20Q LTE-M modem was obtained and a cellular data plan for the

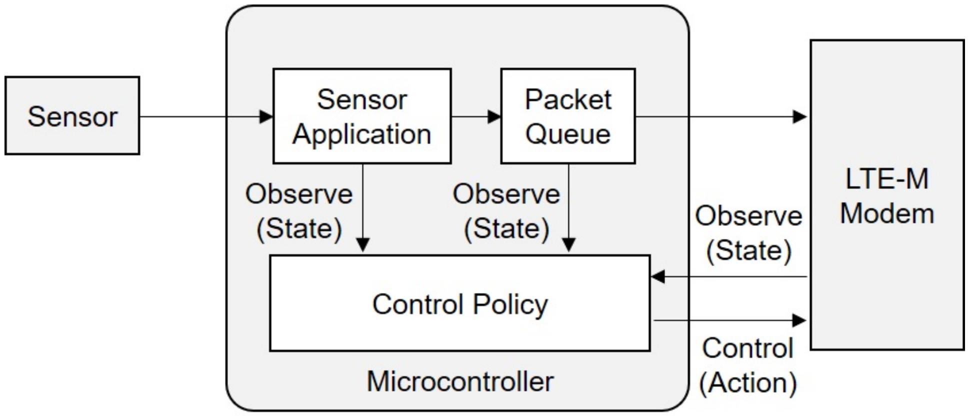

Figure 4.2: Block diagram of wireless sensor CPS.

Verizon Wireless LTE-M network in the United States was purchased. This is a live Internet Protocol (IP) network with nearly nationwide coverage, and effectively provides a mobile Internet connection that can be accessed from almost any location by an MCU. In this work, the focus is on the upstream flow of information. That is, the collection of data through a sensor and the transmission of that sensor data up to a cloud-based server. The resulting information flow within the CPS node is outlined in Figure 4.2, where the arrows interconnecting the blocks represent the flow of information from one component to another.

Due to the LTE-M power profile, a feasible use case for a system like this would be to keep the modem powered off by default, periodically turn it on to exchange data with the Internet and then power it back down to conserve energy. To model this use case, the Sequans Monarch LTE-M modem was controlled from a test application on a laptop computer that powered the modem on, connected to the nearest Verizon Wireless cell tower, transmitted some information (representing a sensor reading) from the test application to a cloud-based server, then disconnected

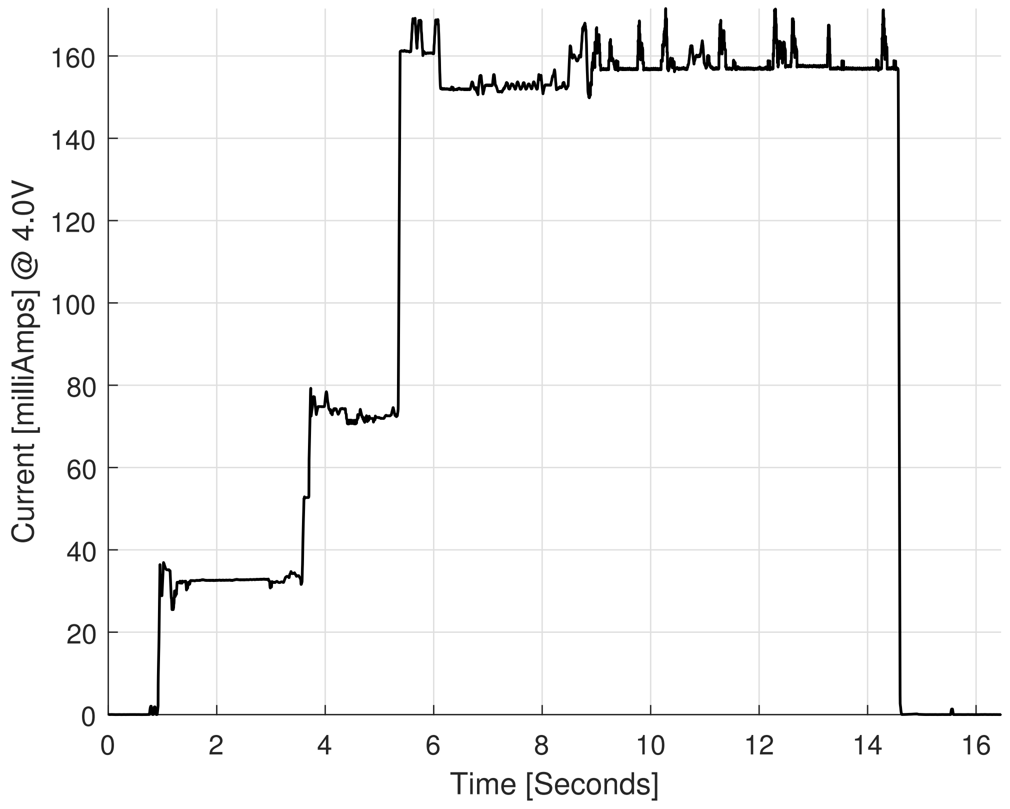

Figure 4.3: Current consumption of LTE-M modem during data transfer.

the cell tower connection, and finally powered down the modem. During multiple runs of this test, the power consumption of the LTE-M modem was measured. A typical current versus time trace of the energy required by the modem to perform this procedure is shown in Figure 4.3. The trace is a time-series of current draw in milliAmperes at a fixed voltage of 4.0 Volts, and thus instantaneous power and total energy for the transaction can both be computed from the data.

Additionally, the quantity of data that is transmitted to the server on each test run was varied. Specifically, a packet size of 256 Bytes per packet was used and the choice of how many of these fixed-sized packets were transmitted was also varied.

These measurements helped to quantify the approximate energy consumption of the LTE-M modem as a function of the amount of data transmitted. This allowed for an approximate model of the modem power consumption to be created, as a function of the quantity of packets transmitted. The derived model is given by Equation 4.4:

$$\begin{gathered} E_{TX}(N_T) = c_1 + c_2(N_T - 1)\,[Joules] \\ N_T \geq 1, c_1 = 6.62, c_2 = 1.55, \end{gathered} \tag{4.4}$$

where E_{TX} is the energy consumption in Joules and N_T is the number of packets. This equation illustrates a defining characteristic of this type of wireless connection: the energy overhead of powering the modem on and connecting to a cell tower can be significantly higher than the incremental cost of transmitting a packet once the connection is active. Thus, if optimizing strictly for energy efficiency it is advantageous to queue up multiple packets before powering on the modem, and then transmit the queued packets together. This approach, however, is the opposite of what should be done if optimizing for transmission latency. This is a fundamental trade-off of controlling the LTE-M modem in this CPS node.

The power control challenge for a system like this consists of implementing an algorithm that strategically determines when to allow new sensor data to accumulate in the sensor, versus when to invoke a communication event (which would produce a power consumption profile similar to the one shown in Figure 4.3) that transfers all packets in the queue up to the cloud-based server. This decision problem involves a trade-off of energy efficiency versus communication latency.

A trivially simple policy is one where any time a new packet arrives in the queue, it is immediately transmitted up to the cloud. In fact, this could be considered a default case where no control policy analysis had occurred, and the MCU program was designed simply to transmit whenever a new packet exists. However, this is still a control policy and is considered as such. This policy will have a low energy efficiency (which is undesirable) and low communication latency (which is desirable).

In another example, a controller could wait until a fixed number of packets accumulate in the queue before doing a batch transfer of all accumulated packets up to the cloud. This policy would have higher energy efficiency but also higher communications latency, as some packets might sit in the queue for some time before being sent to the server. Ultimately, the CPS node's application will dictate what type of latency is desirable for the system, and it should be made as efficient as possible while meeting the specified latency goals.

The policies described above are both examples of very simple "fixed threshold" policies. These are easy to implement and analyze for the dynamics described thus far. An engineer could likely devise other clever policies similar to these, based on heuristics and system simulations. However, in a real-world CPS deployment, the system control dynamics are likely to be more complicated and time-varying than has been described so far. Additional complexity arises from the following phenomena:

- The application is likely to be multi-modal and thus the rate of packet generation within the CPS node is in general time-varying.

- The number of different modes that the application may contain could be much more than two.
- The modem connection time and power consumption are also time-varying due to the physical mobility of the CPS node relative to the fixed location of the cell tower as well as the presence of network congestion from other LTE-M users within the same cell during periods of heavy usage.

When these real-world factors are modeled, it becomes advantageous to consider more involved control policies that can monitor and adapt to the time-varying conditions which may not be precisely modeled or understood prior to a system deployment. For this reason, two approaches are considered that contain runtime learning capabilities specifically to self-optimize continuously at runtime, adapting to the time-varying nature of the CPS node's environment. This feature avoids having to understand and anticipate all of the runtime conditions ahead of time, which for practical purposes is an infeasible requirement for real-world deployments that have very high numbers of nodes.

In Sections 4.6 through 4.8, the multiple options for controlling the CPS node in the case study are described, and then the performance of these policies are compared with regard to the efficiency versus latency trade-off in simulation. Afterwards, the competing options are implemented on a resource constrained MCU and the resulting computational requirements and deployment feasibility are discussed.

4.6 MDP-Based Control

In this section, two approaches to create control policies for the CPS node introduced in Section 4.5 are detailed, and then their performance is compared using simulation. Both approaches use discrete-time MDPs to generate a control policy, which determines when to power the LTE-M modem on and off. The controllers are both designed with the assumption that several of the system's characteristics and dynamics are time-varying and uncontrollable. Thus, both controllers employ some form of learning, in that the policy that each controller employs is continually updated to reflect the time-varying changes in both the system it is controlling and the environment that the system interacts with at any given time. The two controllers can be viewed as being two different realizations of the agent component in the framework of Figure 4.1. These two methods are referred to as: 1) the Structured Learning controller and 2) the Q-Learning controller. The Structured Learning controller is a novel approach described for the first time in this chapter, and the Q-Learning controller is a well known technique in the RL literature (e.g. [58]).

A summary of the differences between the controllers is shown in Table 4.1. The MDP components that are common between the two controllers are: the discrete state space $\mathcal{S}$, the discrete action space $\mathcal{A}$, and the reward function $R(s, a)$. The other components of the controllers are different, namely the STM and policy generation method. The Structured Learning controller contains a parameterized state transition matrix and employs an MDP solver at runtime to generate a con-

	Structured Learning	Q-Learning
State Space	Common: $\mathcal{S}$	
Action Space	Common: $\mathcal{A}$	
Rewards	Common: $R(s, a)$	
STM	Explicitly defined, parameterized	Not needed
Policy Generation	Solver invoked at runtime	Temporal difference equation used to estimate Q function

Table 4.1: Comparison of two MDP-based controllers.

trol policy. In contrast, the Q-Learning controller does not contain an explicit state transition matrix, and instead contains a running estimate of the MDP's Q function. This allows the Q-Learning controller to completely bypass having to store an *STM* and run an MDP solver.

One fundamental difference between the two techniques is that the Structured Learning controller has an *STM* with a pre-populated structure, and only parameters within the structure need to be learned at runtime. There are some aspects of the *STM* that are fixed at design time, and in this way, some structure does not need to be learned. In contrast, the Q-Learning controller has no a priori structural assumptions, and must learn the entire Q function at runtime. Depending on how much of the *STM* is known at design time in a given application, the Q-Learning controller may have to learn more parameters at runtime compared to the Structured

Learning controller.

Structured Learning	Q-Learning
$0 \leq \lvert\Theta\rvert \leq (N_S - 1)N_S N_A$	$N_S N_A$

Table 4.2: Number of parameters to learn at runtime for each controller.

This contrast is shown in Table 4.2, where Θ is the number of unknown or time-varying parameters in the *STM*, as was defined in Equation 4.2. This motivates a general design guideline to be considered: to determine what type of learning approach to use, an engineer should first analyze the percentage of the *STM* that is known versus the percentage that is unknown or expected to be time-varying. In other words, the designer should seek to minimize the number of parameters that need to be learned at runtime, which is $|\Theta|$ in Structured Learning, and $N_S N_A$ in Q-Learning (recall that $|\Sigma|$ denotes the cardinality of a set Σ). If many parameters out of the *STM* need to be learned at runtime, it is possible that $|\Theta| \geq N_S N_A$, in which case the Q-Learning approach would have less parameters to learn at runtime and likely provide better adaptation performance.

The two different controller designs investigated in this section were formulated to concretely explore trade-offs between model-based (Structured Learning) and model-free (Q-Learning) in a tangible, real-world example. In the remainder of this section, the common components of the two techniques are presented first, followed by their differences.

4.6.1 State and Action Spaces

Both controllers utilize a multivariate state space, a concept introduced in Section 2.4.4. The MDP state space is essentially the combination of each of the states of the sensor application, packet queue and LTE-M modem. This is defined as shown in Equation 4.5:

$$\begin{aligned} s = (s_a, s_q, s_m) &\in \mathcal{S} \\ s_a &\in \{0, 1, \dots, N_{SA} - 1\}, \\ s_q &\in \{0, 1, \dots, N_{SQ} - 1\}, \\ s_m &\in \{0, 1, \dots, N_{SM} - 1\}, \end{aligned} \tag{4.5}$$

where $s \in \mathcal{S}$ is the state variable, which is composed of three separate variables s_a, s_q, s_m: the sensor application state, queue state and modem state, respectively.

In general, the interest here is in applications that can run in one of a set of alternative modes (see Section 4.5). Thus s_a is defined as the application mode, a state variable taking a value out of a discrete set of N_{SA} modes. The packet queue has a specific number of packets in it at any given time. In order to allow the controllers to make decisions based on the current state of the queue, s_q is defined as the number of packets in the queue and made part of the MDP state space.

The queue is assumed to be a fixed size $N_{SQ} - 1$, due to the goal of housing it in a resource constrained MCU. The queue can only hold up to $N_{SQ} - 1$ packets, and any attempts to queue additional packets beyond this limit will result in the newest packet being discarded. The discarding of a packet represents a loss of data and is a very undesirable event, that the controllers must seek to avoid through the decisions in their respective control policies.

It should be noted that in some applications, some amount of data loss may be tolerable. Such applications can be accommodated readily in both controller designs by making suitable adaptations. The details of those adaptations are omitted in this chapter for brevity.

The LTE-M modem is a complex mixed-signal System-On-Chip (SoC), containing the LTE-M protocol implementation and runtime signal processing. There is an enormous state space that could be defined for the inner workings of the modem. However, for these controllers the majority of that information is not relevant and thus the modem state is collapsed into a small set of states that is detailed enough for the controller to implement a high performing control policy, without being overly burdened with the computational and storage implications of a large state space. In this spirit, the concept of *transition states* introduced in Section 2.4.3 is utilized.

In this design context, the system being controlled contains many discrete states. Depending on the level of modeling and decision making that is desired, as much as tens of thousands of states or more could be considered relevant. In general, the modeling process involves making design-time decisions as to what level of detail is modeled in the MDP, for each of the system characteristics and dynamics. More fine-grained detail allows for a more precise model, but this leads to a large state space and the computational challenges associated with that.

The concept of transition states allows for significant reduction of the state space to occur when the system passes through a large set of states for a limited time, and the only relevant detail is when the system enters and exits the set as a whole. In some way, these trajectories need to be modeled in the MDP. By utilizing the proposed concept of transition states, a large group of fine-grained states can be abstracted into a single coarse-grained state. A coarse-grained state derived in this way is what is referred to here as a *transition state*.

While the system is actually in one of the fine-grained states abstracted by a transition state τ, the system is modeled using τ as being *in transition* between the predecessor and successor states of τ. The transition state encapsulates a set of discrete states that are present but not relevant to the decision process being designed. The only transition probability needed is derived from an estimate of the expected time until the system leaves the set of states abstracted by τ. Inaccuracies resulting from the associated estimation process represent a potential design trade-off: the creation of transition states out of larger sets of fine-grained states may result in lower accuracy in the overall MDP model. Designers have significant flexibility to control the number and granularity of transition states to help optimize this trade-off.

In this spirit, the large number of fine-grained LTE-M modem states are collapsed into three coarse-grained states: M_OFF, $M\_CONNECTING$, and $M\_CONNECTED$. The $M\_CONNECTING$ state is a transition state, and the other two are not.

M_OFF refers to the modem being fully powered off, and the modem can remain in this state indefinitely until commanded otherwise. $M\_CONNECTING$

is the state that begins immediately after the modem has been powered on and ends when a successful Internet connection has been established. The modem cannot remain indefinitely in this state. By definition, it is guaranteed to transition out of this state after a specified amount of time steps. $M\_CONNECTED$ is the state when a working Internet connection has been established and continues to be maintained. Transmission of packets is only possible in the $M\_CONNECTED$ state. This modeling approach requires the selection of a scalar constant for the modem's power consumption in each of the three states.

As can be observed from Figure 4.3, the power is roughly constant in the M_OFF and $M\_CONNECTED$ states. However, this is not the case in the $M\_CONNECTING$ state. This is addressed by representing power consumption during the entire transition as a fixed value: the average power consumption during the duration of the transition. This simplification is a way of providing the MDP the information that it needs to implement a high performing policy, in as compact a representation as possible. This modeling approach is a design choice, and it is noted that an interesting area for future study is in the trade-offs for varying levels of modeling expressiveness in this area.

In this system, the controllers are being tasked with turning the LTE-M modem on and off. This binary control implies an "on" action that powers on the LTE-M modem and commands it via the modem's interface to attach to the cell tower and establish an Internet connection. Conversely the "off" action implies tearing down any existing Internet connections, and shutting off the LTE-M modem gracefully via the modem's shutdown procedures. The resulting action space is shown in

Equation 4.6. The two actions *off* and *on* in Equation 4.6 are abstractions of multi-step LTE-M modem command sequences.

$$a \in \mathcal{A} = \{\mathit{off}, \mathit{on}\} \tag{4.6}$$

4.6.2 Rewards

In order to "motivate" the controllers to find an effective balance to the latency versus energy efficiency trade-off described in Section 4.5, the reward function is defined as shown in Equation 4.7. The reward function maps each state-action pair (s, a) to a scalar reward.

$$\begin{aligned} R(s,a) = r_1 I(s,a) + r_2 N_T(s,a) + r_3 N_D(s,a), \\ r_1 = -10, \\ r_2 \in \{3, 4, \ldots, 10, 100, 1000\}, \\ r_3 = -100, \end{aligned} \tag{4.7}$$

where $I(s, a)$ is the average electrical current consumed by the modem, $N_T(s, a)$ is the number of packets known to be transmitted, and $N_D(s, a)$ is the number of packets dropped by the modem due to an overflowing queue in the previous timestep. For each of these quantities the function arguments (s, a) are used to denote the respective value of each of the terms known, expected, or averaged when action a is taken in state s. Instead of power consumption, electrical current is used in its place due to it being equally suitable (given a constant voltage) and more straightforward to measure with an MCU in an embedded system.

With this formulation, the scalar reward is thus a linear combination of observable time-varying signals and quantities. This formulation steers both controllers to the desired goals, by rewarding them (with a positive reward value) when a packet is transmitted successfully and penalizing them (with a negative reward value) when electrical current is consumed or the packet queue overflows. The reward constants r_1, r_3 were selected via experimentation and r_2 was left as a free parameter in order to be able to generate a set of instances for each controller. Each instance in the set places different amounts of importance on the latency requirement relative to the energy efficiency requirement. This approach allows for simulation of a suite of controllers for each method, and plotting the resulting performance for a more robust comparison. The resulting policies are ones where the respective controllers turn the modem on and off at each discrete time step, in the way they determine is the optimal approach for obtaining the maximal rewards. In other words, they attempt to transmit the packets generated by the sensor application without incurring undesired delay or consuming more electrical power than is needed, through dynamic and changing conditions.

In Section 4.6.3 through Section 4.6.4, the differences between the two controllers that being evaluated in this case study are detailed.

4.6.3 Structured Learning Controller

The Structured Learning controller consists of the common components described above, plus the addition of the *STM*s and an MDP solver. The *STM*s are described in this section, and the solver is described in Section 4.8.

The stored *STM*s at any given time are a combination of constants and time-varying parameter estimates. The constants are programmed in at design time, and the estimates are maintained by observing samples of the relevant quantities, and using the Temporal Difference (Equations 4.3) method to update the estimates. The estimates are plugged into the *STM* data structures, which serve to maintain fully populated *STM*s at each time step.

The *STM*s are constructed using a factored formulation, which greatly reduces the storage requirements of the MDP. The factorization procedure is shown in Equation 4.8 through Equation 4.12. The factorization serves to convert one large multivariate conditional probability distribution into a product of functions, each having the form of a lower dimensional conditional probability distribution. The terms correspond to the subsystems of the sensor application, packet queue and LTE-M modem, respectively. This re-arrangement enables a significant reduction of the MDP storage requirements.

$$p(s^{(n+1)}|s^{(n)}, a^{(n)}) = p(s_a^{(n+1)}, s_q^{(n+1)}, s_m^{(n+1)}|s^{(n)}, a^{(n)}) \tag{4.8}$$

$$= p(s_a^{(n+1)}|s_q^{(n+1)}, s_m^{(n+1)}, s^{(n)}, a^{(n)}) \cdot p(s_q^{(n+1)}, s_m^{(n+1)}|s^{(n)}, a^{(n)}) \tag{4.9}$$

$$= p(s_a^{(n+1)}|s_a^{(n)}) \cdot p(s_q^{(n+1)}, s_m^{(n+1)}|s^{(n)}, a^{(n)}) \tag{4.10}$$

$$= p(s_a^{(n+1)}|s_a^{(n)}) \cdot p(s_q^{(n+1)}|s_m^{(n+1)}, s^{(n)}, a^{(n)}) \cdot p(s_m^{(n+1)}|s^{(n)}, a^{(n)}) \tag{4.11}$$

$$= p(s_a^{(n+1)}|s_a^{(n)}) \cdot p(s_q^{(n+1)}|s^{(n)}, a^{(n)}) \cdot p(s_m^{(n+1)}|s_m^{(n)}, a^{(n)}) \tag{4.12}$$

4.6.3.1 Sensing Application

Given the observability of the sensing application's mode, the controller can maintain parameters that statistically characterize how often the application is in a given mode, and how likely the CPS is to transition from any mode to any other given mode. These characterization parameters are listed in Equation 4.13. Using the latest values of these parameters, the s_a term of the factored STM can be fully instantiated.

$$\hat{\sigma}_{i,j} = p(s_a^{(n+1)} = j|s_a^{(n)} = i)\forall(i,j) \in \{0, 1, \ldots, N_{SA} - 1\}^2 \tag{4.13}$$

4.6.3.2 Packet Queue

Since it is part of the state space, the dynamics of the queue must also be modeled as a transition matrix. This results in having to model a fully deterministic process into stochastic structures in order to fit into the MDP framework, and thus most of the probabilities are either 1 or 0. The transition probabilities are almost all known at design time, since the dynamics of the queue do not change. The only uncertainty comes from the rate of packets entering the queue from the sensing application, and the rate leaving the queue from the LTE-M modem connection.

The packet queue's state is defined as the number of packets in it at a specific timestep. The transition probabilities amount to the likelihood of transition to another state, in other words the change in the number of packets. The transition to a state where more packets are inserted corresponds directly to the sensing application's packet generation rate. These events are combined with the probability of packets being removed from the queue by being transmitted to the cloud server. If the modem state and action are such that the modem is not yet connected, then no packets can leave the queue and thus transitions to states where the number of packets is reduced are not possible. If the modem is connected, then packets can leave the queue.

4.6.3.3 LTE-M Modem

In the model, the dynamics of the LTE-M modem are a direct application of the transition state concept. In order to instantiate this component of the STMs, the controller needs to maintain a running estimate of how long the LTE-M modem takes to connect to the network. This time-varying quantity is referred to as T_C and its most recent estimate as $\hat{T}_C$. With this estimate, the resulting transition probabilities are shown in Figure 4.4. Following the transition states theory introduced in Section 2.4.3), the value of the parameter ρ is defined in Equation 4.14, where

	OFF	CONNECTING	CONNECTED
OFF	1	0	0
CONNECTING	1	0	0
CONNECTED	1	0	0

	OFF	CONNECTING	CONNECTED
OFF	0	1	0
CONNECTING	0	1-ρ	ρ
CONNECTED	0	0	1

Figure 4.4: State transition matrices for LTE-M Modem.

T_F is the duration of one control frame.

As can be observed from the Figure 4.4 and Equation 4.14, this component of the STMs is parameterized by the running estimate $\hat{T}_C$ (through ρ) in two of the matrix elements, and combined with known constants for the remaining matrix elements.

$$\hat{\rho} = \left[\mathit{floor} \left(\frac{\hat{T}_C}{T_F} \right) \right]^{-1} \tag{4.14}$$

4.6.4 Q-Learning Controller

The Q-Learning controller was implemented directly from the description of the technique in [58]. In this method, a function Q is created as a mapping $Q(s,a) : (\mathcal{S} \times \mathcal{A}) \rightarrow \mathbb{R}$, where $\mathbb{R}$ denotes the set of real numbers. Each mapping in the function represents an estimate of the total amount of reward an agent can expect to accumulate over the future, starting from a given state s and taking a given action a. The Q function is updated on each iteration of the controller using the temporal difference equation. Using the latest estimated version of $Q(s, a)$, the action is selected by comparing all actions for the given state s and selecting the action with the largest value. The remaining details of the Q-Learning method can be found in [58].

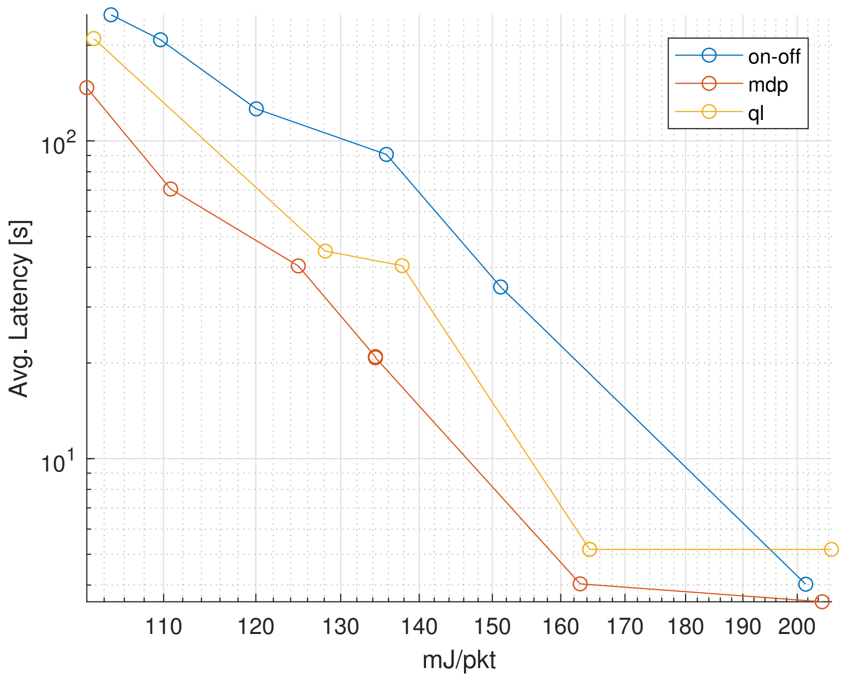

Figure 4.5: Simulation Results: Energy efficiency versus communication latency.

4.7 Simulation

In order to objectively compare the runtime performance of the Structured Learning and Q-Learning controllers presented in Section 4.6, a MATLAB simulation was created containing models of all the subsystems described in the case study. In the simulation testbed, a sensor application generates packets at rates consistent with a given mode, and also simulates the transition between modes at specified transition rates. A packet queue object models a generic fixed length queue, which acts as a First In First Out (FIFO) data structure, and overflows if the maximum number of elements is exceeded. A dynamic model of the LTE-M modem was created using the collected time-series data and electrical power measurements (see Section 4.5.1).

The simulation was run with three separate controllers: the two MDP-based controllers described in the previous section, as well as a third manually-generated policy. The manually-generated policy simply checks the queue, and turns on the modem any time a specified number N_q of packets are in the queue, where N_q is a parameter of the policy. Once the modem is turned on, it remains on until the queue is empty.

The simulation was run multiple times for each controller, and each simulation run is represented by a single point in the graph of Figure 4.5. The fixed threshold (manually generated), Structured Learning, and Q-Learning approaches are denoted by the "on—off", "mdp", and "ql" traces, respectively. The best performance corresponds to points that have the lowest average communication latency (the vertical axis) and simultaneously the lowest average energy efficiency (the horizontal axis).

For the fixed threshold technique, multiple policies were generated by varying the N_q threshold at which the modem was powered on. For the MDP-based controllers, the r_2 constant in the reward function from Equation 4.7 was varied. This approach produced a set of control policies for each controller, allowing a full exploration of the performance limits of each technique.

The first conclusion that can be made from the data in Figure 4.5 is that both MDP-based controllers outperform the fixed threshold approach, for all possible values of the N_q threshold. This is likely a result of the MDP-based policies being richer and more expressive; while the fixed threshold policies are a function of the packet queue state only (ignoring the application and modem characteristics), the MDP-based policies materialized as a non-trivial function of the entire state space.

In this case study, since the MDP is able to reason using algorithmic methods on data structures and computations on conditional probabilities, it can consider more effects and consequences systematically and produce highly optimized policies that are more expressive relative to the simple, manually-derived, fixed threshold heuristic.

The second conclusion that can be made from the data is that the Structured Learning controller outperforms the Q-Learning controller. As an example, if the rewards are tuned such that both learning controllers achieve an average packet transmission latency of 20 seconds, the Structured Learning controller is able to accomplish this with an average energy efficiency of 135 mJ per packet, compared to 163 mJ per packet on the Q-Learning controller. This amounts to a 17% savings in the transmission energy for sending the exact same packets at the same average latency. This can be an important difference since transmission energy is often the largest source of energy consumption on an LTE-M connected sensor.

This improvement is a direct consequence of the Structured Learning algorithm focusing the learning on only the time-varying aspects of the system (e.g. modem power, cell tower connection time, etc.). and accepting as unquestionable truth the other dynamics and attributes (e.g. that the packet queue contains one less packet after a packet is removed from it, etc.). In contrast, the Q-Learning controller is forced to learn (and continue to update indefinitely) all aspects of the system transition probabilities in response to selected actions. It must continually experiment with exploratory actions and accumulate data to learn all of the system dynamics (including well understood behavior, such as the packet queue's dynamics). These results are consistent with the claim that expending learning effort on such immutable aspects is both unnecessary and detrimental to the overall system performance.

4.8 Implementation

This section details the results of implementation experiments performed to assess the viability of the competing MDP-based control strategies in the context of a state-of-the-art processing platform for resource constrained CPSs. The alternatives are implemented on a typical MCU that would be used to realize the CPS case study, and are compared in terms of their execution time, memory usage and processing power consumption.

4.8.1 Experimental Setup

The competing controllers were implemented on the Silicon Labs EFM32GG, a small and low power ARM Cortex M3-based MCU. The processor was running on the EFM32 STK3700 development kit, which houses the CPU as well as sophisticated energy monitoring circuitry. The EFM32GG contains 128 kB of RAM and 1MB of FLASH, which make it a typical example of a resource constrained platform for the CPS in the case study at the time of this writing.

In order to compare the controllers objectively, the following experimental setup was created on the EFM32GG development board. All controllers were implemented in C and stored in the MCU's program memory one at a time. Memory usage was computed by statically allocating all data structures and examining the map file that the MCU's compiler generates. A common test harness was written for the EFM32GG, which was driven by a periodic timer interrupt. The interrupt rate was configured to be 100ms, which was used as the fixed-period discrete-time iteration rate for the controllers.

The C program initially puts the CPU into its low power sleep mode. It remains in that mode until the periodic interrupt fires. Once the interrupt fires, the MCU is woken from sleep and it then executes the computations needed for one iteration of the controller under test. Once the iteration has completed, the MCU returns back to sleep mode where it waits for the next firing of the periodic interrupt. Since the sleep current is extremely low compared to the run current (microAmperes compared to milliAmperes), this approach enabled precise measurement of both the execution time and computational energy required to execute each controller on the MCU by observing the current versus time profile of each controller. Real-time MCU current consumption was measured by using the EFM32GG board's energy monitoring tools, which allow very accurate current versus time data to be observed in the form of a high resolution time-series waveform capture.

Using this simple fixed rate scheduling scheme combined with the Cortex-M3 sleep modes and the development board's current monitoring tools, it was possible to observe the execution time and processing current consumed by the CPU for each control policy. This testbench provided a highly repeatable experimental setup where all settings were kept the same from case to case with the only difference being the control policy being used.

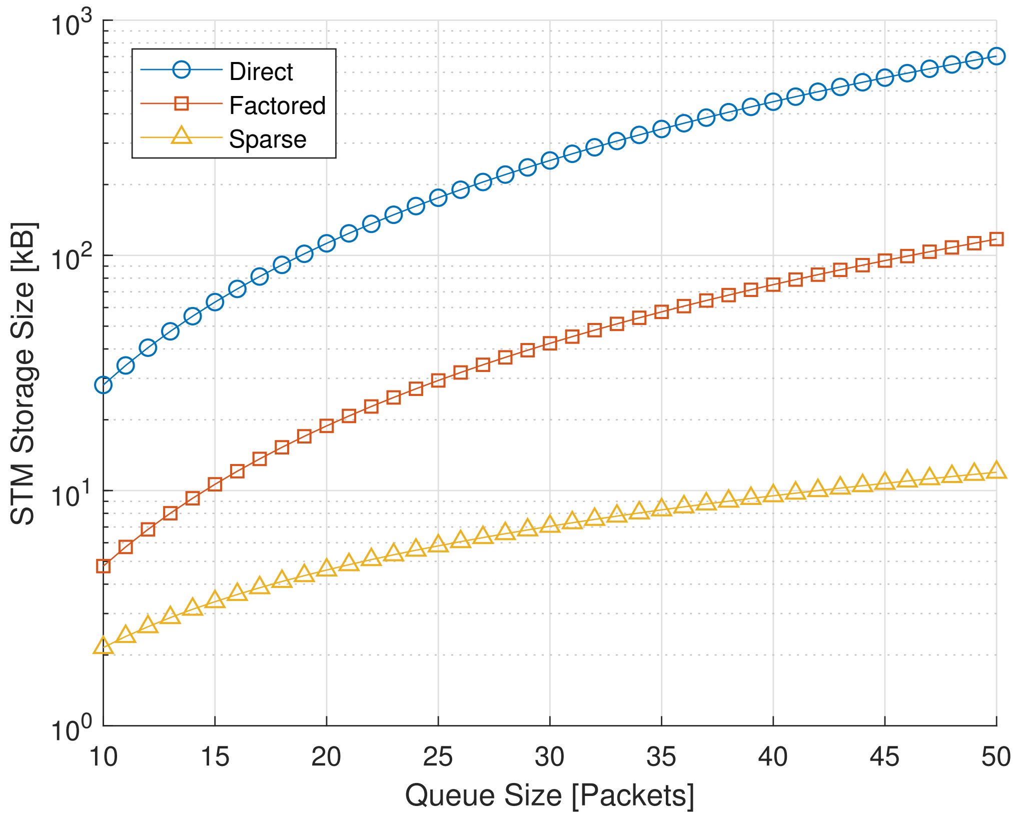

Figure 4.6: Transition matrix storage sizes.

4.8.2 Matrix Format

In the case study's MDP, the number of states N_S is 66, and the number of actions N_A is 2. The number of non-zero elements K_{NZ} in the STMs is 444. This represents a sparsity of 94.9%. Aside from the direct implementation of the full STMs, two CMM techniques were evaluated: a factored implementation and a sparse implementation. Figure 4.6 shows the resulting STM storage sizes for these techniques on the case study, over a range of packet queue sizes.

As can be observed from the data, both the factored and sparse implementations reduce storage size considerably. However, the sparse method is the most effective in this regard and for this reason it was selected as the approach for the implementation. The implementation used a single byte to store each element of the policy lookup table, and four bytes to store the floating point elements of the STMs and reward functions.

4.8.3 Measurements

4.8.3.1 Memory Usage

First, the techniques are compared in terms of how much data storage each required on the MCU. The results are shown in Table 4.3, where the column labeled Sparse Structured Learning represents the Structured Learning method implemented with sparse matrices, as described in Section 4.4.3.

The results in Table 4.3 show that the Q-Learning approach is the most favorable in this metric, and furthermore show that it requires significantly less data stor-

	Structured Learning (VI)	Sparse Structured Learning (SVI)	Q-Learning
STM	34.0 kB	2.60 kB	-
Rewards	0.51 kB	0.51 kB	0.51 kB
Q function	-	-	528 B
Total	34.5 kB	3.11 kB	1.03 kB

Table 4.3: Data storage size in kiloBytes.

age than the Structured Learning (VI) approach. However, when the SVI method is applied to Structured Learning (SVI), there is a significant reduction in the required data storage. The data shows that in this case study, the CMM techniques reduce the storage requirements of Structured Learning to be much closer to that of Q-Learning, while not beating it in this regard.

Generalizing these results beyond the case study, it can be seen from Table 4.3 that all of the methods being compared require the same amount of storage for the reward function. Thus, the difference in memory usage is attributed to the storage of the STMs in the VI and SVI controllers, compared to only the Q function in the Q-Learning controller. This difference can be calculated in the general case as follows. Assuming STM entries are 4 byte single precision floating point values, the Q function can be stored in $4N_SN_A$ bytes, as it consists of a table of N_SN_A floating point values. The storage size of the STMs in the Structured Learning (VI) method is $4{N_S}^2N_A$ bytes, as it consists of N_A stochastic matrices, each of size $(N_S \times N_S)$.

The STMs in the Sparse Structured Learning (SVI) method were implemented with sparse matrices stored in coordinate format [43]. This format stores only the non-zero elements of a matrix, along with two indices representing the column and row index of the element, respectively. Assuming that $\lceil log_2(N)/8 \rceil$ bytes are required to store an integer index that can take on one of N values, the storage size required for the STMs in coordinate format is shown in Equation 4.15, where K_{NZ} is the number of non-zero elements in the STMs.

$$K_{NZ}(\lceil log_2(NsNa)/8 \rceil + \lceil log_2(Ns)/8 \rceil + 4) \tag{4.15}$$

Evaluating the formula above (Equation 4.15) using the constants from the case study ($N_S = 66$, $N_A = 2$, $K_{NZ} = 444$) results in the storage sizes shown in Table 4.3. The formula can be used to predict the required storage sizes for other case studies by appropriately changing the values of (N_S, N_A, K_{NZ}).

4.8.3.2 Computation

	Structured Learning (VI)	Sparse Structured Learning (SVI)	Q-Learning
Per Control Iteration:	3.5 μs / 117 nJ	3.5 μs / 117 nJ	211 μs / 7.06 μJ
Per Solver Iteration:	50.1 s / 1.67 J	5.61 s / 187 mJ	-
Average Power:	483 μW	69.8 μW	78.8 μW

Table 4.4: Execution time (in seconds), computation energy (in Joules) and average power (in Watts).

Next, the execution time and power consumption of the MCU were measured, when executing each of the competing techniques. The results are shown in Table 4.4. These results show that Q-Learning requires the same computation on every control period. The control operation consists of updating the Q function based on the observed state transition and reward, and computing the best action for a given state using the latest values in the Q function. The Q-Learning method performs these operations on every control period.

In contrast, the Structured Learning techniques (VI and SVI) run a solver, which was invoked once per hour. These techniques compute a new control policy every hour, and the time and energy required to do this is shown in the second row of Table 4.4. After computing a new policy every hour, the Structured Learning techniques simply look up which action to use from a stored table for the remainder of that hour.

The data in Table 4.4 shows that the Structured Learning techniques involve much less computation time and energy consumption during a typical control period relative to Q-Learning. Note that the first row of data for Structured Learning in Table 4.4 excludes the computation associated with solver execution, while the third row of data (labeled Average Power) includes the effects of solver computation.

It is important to note that the Structured Learning implementation would likely require a priority-based preemptive scheduling scheme such that the control iteration execution would take higher execution priority over the solver, such that any real-time deadlines associated with the controller are not missed due to running the solver.

The data in Table 4.4 shows that although Q-Learning does consume less average power than Structured Learning (VI), when the SVI method is applied to Structured Learning the average power is reduced lower than Q-Learning. SVI reduces computation time by replacing standard matrix operations by sparse matrix operations. This results in a significant reduction in computation time, given that the *STM*s are extremely sparse.

Generalizing these results beyond the case study, it is possible to identify the factors that affect which of the methods is favorable in terms of computation costs — e.g., processing time and energy consumption. In the Structured Learning techniques, the size of the MDP and complexity of the *STM*s determine how long it takes to execute a solver to produce a control policy. If the MDP is very large and complex, the solver will take longer to execute. In contrast, the Q-Learning technique is not affected at all by this attribute. In this aspect, it can be concluded that Q-Learning is better suited to deal with large MDPs than Structured Learning in terms of computation expense.

Another factor that is relevant is how often the Structured Learning techniques are required to compute an updated policy. In general, a suitable update rate is determined by the application's adaptation requirements, and how quickly the time-varying environment is changing. In the case study presented here, the policy is updated once per hour, but a system that adapts to more slowly changing dynamics may only need to update the control policy once per day or even less frequently. On the other hand, a system adapting to fast changing dynamics may need to update the policy much more often, such as once per second. A faster update rate will generally

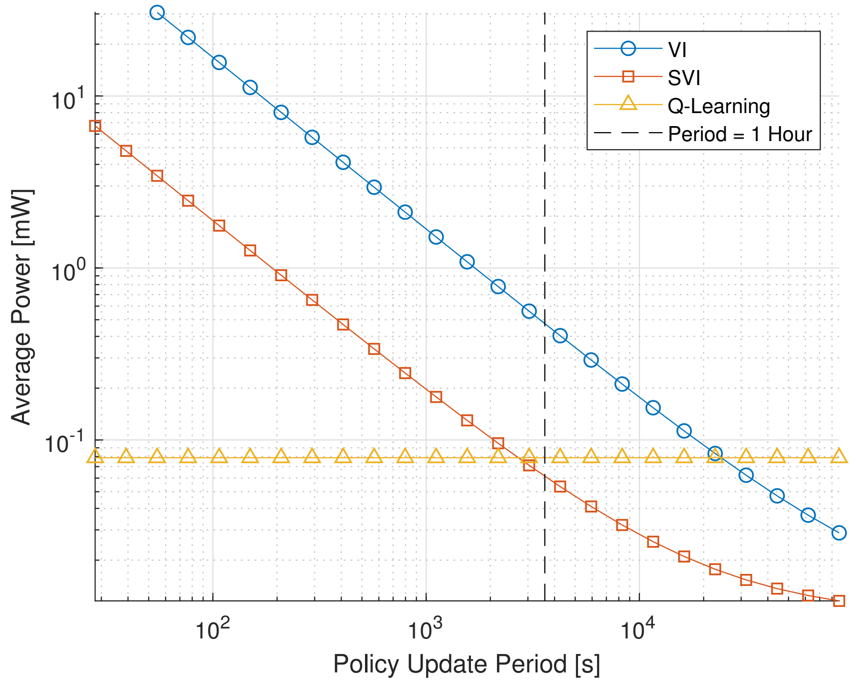

Figure 4.7: Average power consumed by the MCU on competing control algorithms as a function of the policy update period.

increase the computational cost of the Structured Learning techniques, whereas Q-Learning is not affected by this consideration at all. In this regard, Structured Learning is better suited to applications were the adaptation is on dynamics that are varying relatively slowly in time.

In this case study, the data point of a 1 hour update period leads to Sparse Structured Learning having lower computational cost than Q-Learning. The crossover point where Q-Learning consumes less computational power is shown in Figure 4.7 to be approximately at 45 minutes. It is important to note that this crossover point is specific to the MDP size and complexity (affecting solver execution time) and the choice of MCU (affecting run current, sleep current, and solver execution time).

4.8.4 Adaptation Overhead

The improvement in energy efficiency shown in Figure 4.5 is the result of the runtime adaptation making dynamic modem actuation decisions that result in improved performance trade-offs. However, these improvements are obtained at the cost of expending processing energy running the adaptation algorithm, as shown in Table 4.4. An objective analysis of the overall benefits of the runtime adaptation can only be made if the positive benefits of the adaptation are combined with its negative effects.

	Fixed Threshold	VI	SVI	Q-Learning
Energy Efficiency [mJ/packet]	151	125	125	139
Average Modem Power [mW]	1.49	1.24	1.24	1.38
Adaptation Overhead [μW]	-	483	69.8	78.8
Total Power [mW]	1.49	1.72	1.31	1.45
Overall Improvement	-	-15.4 %	12.1 %	2.7 %

Table 4.5: Overall improvement from runtime adaptation.

The numerical result of such an analysis is shown in Table 4.5. The first row is the energy efficiency achieved by each competing method, obtained by selecting an operating point for each method from Figure 4.5. In order for the selections to be a fair comparison, the operating points are chosen to have roughly equal performance in the other performance metric defined for the application — transmission latency. All three selections have an average latency of approximately 35 seconds. The second row is obtained by multiplying the energy efficiency (in milliJoules per packet) times the average packet generation rate in the simulation (in packets per second), which was 0.0099. Thus, the second row represents the average power consumed by the LTE-M modem, when under the control of each competing adaptation method under identical packet generation statistics.

The third row gives the energy expended by running the adaptation algorithm, and is taken directly from Table 4.4. This represents the cost at which the modem energy efficiency improvements are obtained. The fourth row is the sum of the second and third rows, and represents the total power that is consumed by the sensor node. The last row quantifies the percent improvement over the Fixed Threshold case, which represents a baseline in which no online adaptation method is used.

The numbers in the table show that the best overall option is Structured Learning using the SVI algorithm, with a 12.1% improvement in power consumption over the baseline, when the other metric — transmission latency — is held roughly constant across the alternatives. Additionally, it can be seen that although Q-Learning runs at a slightly lower computational cost, the adaptation performance in terms of energy efficiency is not as good, leading to a worse overall performance.

A final observation that can be made from this data is that Structured Learning provides superior modem efficiency gains with both the VI and SVI algorithms, however under the VI algorithm those gains are effectively lost due to the relatively large processing power required to run the VI algorithm. The overall improvement of this configuration is -15.4%, meaning it actually makes the performance worse than not having any adaptation at all. It is only through the use of the much more power-efficient SVI algorithm that Structured Learning becomes an attractive choice, and in fact even outperforms Q-Learning in this regard.

4.9 Conclusion

In this chapter, a survey of recent developments in Compact MDP Models (CMMs) was provided, and by integrating several complementary CMM techniques, a novel CMM-based approach to CPS design was developed. Comparisons between CMM-based methods and Q-Learning in the context of CPS were made. The differences between the two approaches were explored conceptually, as well as through a detailed case study involving both simulation and a prototype implementation.

From the results of this chapter, it can be concluded that Q-Learning can be considered a more robust technique when either very little is known about the system a priori, or a large percentage of the dynamics are expected to continually change at runtime. In contrast, when a significant portion of the system's environment or its dynamics are fixed, the CMM option can provide a more efficient and robust approach.

An LTE-M connected sensor was detailed as a CPS case study to compare a CMM-based learning controller to an alternative controller that used Q-Learning. For a specified average packet transmission latency, the CMM-based controller resulted in a 17% reduction in LTE-M transmission energy, which is often the largest source of energy consumption on an LTE-M connected sensor. The energy savings are accomplished through strategic management of the LTE-M modem and connection status, using learned dynamics of the system and its environment.

Since the learning controller must be implemented in the deployed system, and its processing can be considered an overhead to the LTE-M connected sensor's main purpose, the implementation costs of the two learning controllers were also analyzed. The implementation was on a small microcontroller that is typical of what would be used for such a CPS system. In this experiment, it was found that the CMM-based controller used 69.8 μW compared to 78.8 μW for the Q-Learning controller, an 11.4% savings. However, the CMM-based controller required more RAM to store its data structures — 3.11kB compared to 1.03kB.

Finally, the overall benefits of using runtime adaptation were analyzed by comparing the improvement of the LTE-M transmission energy alongside the processing overhead of running the adaptation algorithm. In this analysis, Structured Learning was found to outperform Q-Learning only in the case when a power efficient implementation of the algorithm is used. This result concretely demonstrates the need for runtime adaptation techniques to be very lightweight in the overhead they add to the system in terms of memory and processing power consumption. If the adaptation techniques are too resource intensive, they can cancel out the energy efficiency gains made by the runtime adaptation, and even lead to worse overall performance than executing the system without any adaptation.

Useful directions for future work include explorations into other challenging CPS case studies with larger state spaces, and continued development of compact techniques that provide self-awareness and runtime adaptation capabilities at all levels of embedded implementation.

Chapter 5

GEMBench: A Platform for Collaborative Development of GPU Accelerated Embedded Markov Decision Systems

5.1 Introduction

The previous chapters have shown the use of MDPs as increasingly relevant tools in the design of Embedded Computing Systems (ECSs). However, progress in this area currently suffers from a lack of common benchmarking methodologies. The work presented in this chapter helps to bridge this gap. Material in this chapter was published in preliminary form in [67].

More specifically, this chapter presents a summary of challenges associated with MDP-based design for ECSs, a survey of the state-of-the-art in MDP solvers and datasets that are relevant to embedded systems, and a novel open source software package for facilitating experimental research in the implementation and application of embedded MDPs.

In recent years, a new class of MDP solver implementations has emerged that uses GPUs for acceleration. Examples of solvers in this class include those presented by Noer [68], Ruiz and Hernandez [42], and the SPVI algorithm introduced in Chapter 3. The results in these works show performance improvements of roughly an order of magnitude beyond what is possible with CPU-only solvers. Additionally, in Chapter 3 a variable dependency analysis has been presented to provide insight into why MDP solvers can benefit significantly from the parallelism available in GPUs.

Motivated by the increasing interest in deploying MDPs within resource constrained embedded systems, and the promising role of GPUs to support such deployments, this chapter presents a novel open source platform for testing and benchmarking GPU-accelerated ECSs that employ MDPs. This platform is referred to as GEMBench, which stands for the Gpu accelerated Embedded Mdp testBench.

GEMBench is designed to help researchers address significant logistical challenges in incorporating MDPs and their solvers into novel embedded system designs. An important decision point in this context is whether to develop a new MDP solver or to use an existing implementation. This decision point leads naturally to the following questions: Which existing open source MDP solver alternative is the best to use or compare against for a given set of system design constraints? How much processing time, memory and power does a given solver consume in order to solve a given MDP? Can an MDP solver's performance be improved through optimizations or algorithmic innovations?

GEMBench is targeted to a specific embedded GPU platform, the NVIDIA Jetson platform, and is designed for future retargetability to other platforms. The orientation to a specific platform is important for the objectives of GEMBench, which include promoting quantitative comparison among alternative MDP solvers and implementations.

Additionally, an open source software package [69] is contributed, called the *GEMBench Package*. The GEMBench Package can be downloaded onto the targeted platform to create a development testbench. The testbench contains implementations of published solvers, datasets to run the solvers on, reference solutions to the datasets, documentation on how to measure relevant performance metrics, and guidance on how to contribute future developments to the framework in a consistent manner. The testbench, which encompasses the GEMBench-compatible

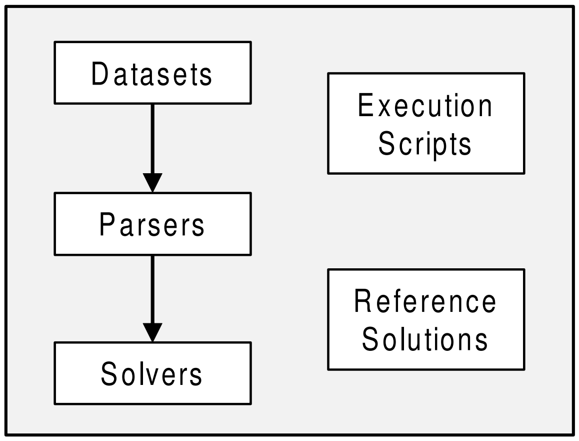

Figure 5.1: Block diagram of the GEMBench Package.

platform (NVIDIA Jetson) and the GEMBench Package, is what is referred to here as *GEMBench*.

A block diagram of the GEMBench Package, illustrating the major components is shown in Figure 5.1. The arrows denote the flow of information, where, in a given experiment, a selected solver utilizes a selected MDP format-specification parser to solve a selected MDP. The solver, parser, and MDP are selected from three extensible libraries, respectively. The package is intended to allow new solvers to be written by researchers, who would immediately have datasets to run them on and reference solutions to compare them with in order to validate correctness. Additionally, performance measurements, such as execution time and power consumption, can be obtained and compared to other reference solvers provided by the package. Furthermore, all of this can be automated to efficiently create extensive benchmarking data through the use of execution scripts, which are also included with the package.

The remainder of this chapter is organized as follows. In Section 5.2, a summary is presented of the current landscape in MDP solvers, common datasets and platforms. In Section 5.3, GEMBench and its associated software package are described in detail. In Section 5.4, the results for representative experiments using GEMBench are provided. Finally, Section 5.5 concludes with interesting directions for future work.

5.2 Survey

This section presents the results of a survey of existing MDP solvers, reference platforms and benchmarking efforts to date. The survey focuses on aspects of MDP solvers that are important to understand from the viewpoint of experimenting with and deploying them on embedded systems.

This survey is a subset of a much larger body of work. There are dozens of papers in the literature documenting MDP solver algorithms or techniques. Out of those efforts, only a subset contain an attached or referenced software implementation of the technique. It is on those works with a corresponding software implementation that are the primary focus here.

5.2.1 Solver Implementations

Table 5.1 and Table 5.2 summarize the CPU-based and GPU-based MDP solver implementations that were found to date, respectively. From this survey, one can conclude first that MDP solvers have been an active area of research and development for the last 25 years. Second, researchers have commonly contributed their own solver implementations to the growing body of work.

Based on these survey results, attempts to determine which of these implementations could be labeled as the current state-of-the-art in solvers were unsuccessful. Several types of complications arose. In some cases, algorithms did not compile on the available computing system. Version incompatibilities with dependencies were suspected in some cases, and poor documentation of multi-step build sequences in

Solver Name	Development Period	Language
pomdp-solve [70]	1994-2007	C/C++
Symbolic HSVI [71]	1998-Today	Java, Perl
MDP Toolbox for MATLAB [72]	1999-2002	MATLAB
ZMDP [73]	2004-2016	C/C++
SPUDD [47]	2007-2011	C/C++
Symbolic Perseus [74]	2007-2009	MATLAB, Java
APPL [56]	2009-2017	C/C++
C++ MDP Solver [75]	2010-2018	C/C++
MDPSOLVE [33]	2011-2015	MATLAB
libpomdp [76]	2011-2014	MATLAB, Java
AI-Toolbox [77]	2015-Today	C/C++

Table 5.1: CPU-based MDP solver implementations.

other cases. In other cases, solvers compiled and ran successfully, but were not compatible with the same MDP files as other solvers. This is a critical shortcoming that effectively prevented objective comparison (without laborious conversion between different file formats).

5.2.2 Datasets

Benchmarking a solver requires not only a working solver implementation, but also an MDP for the solver to solve. A survey of MDPs used in solver implementations was performed, and the results are summarized in Table 5.3.

Solver Name	Development Period	Language
Noer13 [68]	2013	C++/CUDA
Thrust-VI [42]	2015	C++/CUDA
SPVI [38]	2015-2018	C++/CUDA

Table 5.2: GPU-based MDP solver implementations.

In some cases, researchers simply created an MDP by instantiating the MDP data structures as constants in source code. This approach is perhaps the easiest route to solve a single MDP, but does not scale well to solve many different MDPs. In other cases, researchers stored an MDP's data structures in a file, and then created a means to read in and parse the file in order to provide the MDP to the solver algorithm. Some researchers defined new file formats for representing MDPs, while others adopted formats introduced by previous researchers. Clearly, the use of file-based specification of MDP data structures is more flexible than coding constants into source code, as it allows for one solver to solve many MDPs simply by changing the MDP file. Additionally, this method facilitates sharing of specific MDPs across research groups.

The first standard MDP file format that gained a foothold in this area was the `.pomdp` format [70], sometimes referred to as the Cassandra format (after the last name of the author). This is a human readable text file that describes the MDP components using a custom syntax that seems to have been created precisely for this purpose. The format is well documented, and many example MDPs can be found in this format throughout the literature. In [68], researchers contribute a MATLAB parser for the .pomdp format, which ingests .pomdp files and instantiates an MDP in the MATLAB workspace.

The strength of the `.pomdp` format is that it is easy to read and straightforward to inspect visually. The downside is that it is inefficient in terms of file size. In one case [73], an 180MB file was required to specify an MDP. In another case, in an experiment in the work of Section 2.4 an MDP went above 750MB when stored in `.pomdp` format. In practical use, hundreds of MBs is likely too large of a storage size requirement in general for ECSs, at the time of this writing.

To circumvent the logistical issues of transporting files of this size from one system to another, occasionally researchers have resorted to writing scripts that generate a `.pomdp` file immediately prior to solving it, rather than copying a large `.pomdp` file from one benchmarking setup to another.

In general, it was found that although MDP data structures can be very large, their information content is relatively low compared to their size. As a result, their associated information is highly compressible. See Section 3.4.1 for elaboration on these findings. The findings support the claim that the `.pomdp` format can introduce needlessly high files storage requirements.

The second format that has been used across research groups is the `.spudd` format, first created for use by the SPUDD solver [47]. This format is also a human-readable text file, but the MDP format is in the form of tree-shaped data structures known as Algebraic Decision Diagrams (ADDs). ADDs are used in `.spudd` files because the SPUDD solver operates Factored MDPs, as described in Section 2.4.4.

Factored MDPs can often be stored very compactly using tree-shaped structures, and SPUDD even operates on the MDP using tree-shaped structures for the intermediate solver calculations. Other solvers inspired by or derived from SPUDD also use ADDs and `.spudd` files. The downside to this format is that an unfactored MDP must be factored before being stored in this format, and this factorization process can be very difficult or even impossible if the MDP does not contain a specific underlying conditional dependence property. Another downside to this format is that it is difficult to parse (compared to the `.pomdp` format), due to its use of tree-shaped structures and Lisp syntax.

A third effort at defining MDP file formats arose from a series of MDP solving competitions held as part of the International Conference on AI Planning and Scheduling (AIPS), which later merged with the International Conference on Automated Planning and Scheduling (ICAPS). In these conferences, MDP solver competitions were held on 10 occasions from 1998 through 2018. These competitions defined their own file formats, documented them for use by competitors, and provided the MDP files in those formats.

The file formats steadily evolved over the years, including the Planning Domain Definition Language (PDDL) [78] version 1.0, PDDL version 2.1, the Probabilistic Planning Domain Definition Language (PPDDL) [79] version 1.0, and the Relational Dynamic Influence Diagram Language (RDDL) [80]. These formats are by far the most complex (compared with `.spudd` and `.pomdp`), but also the most powerful and expressive. These file formats are designed to specify many different classes of planning and decision problems, beyond just MDPs and POMDPs.

5.2.3 Benchmarking Platforms

In the documents associated with each of the solver implementations that were found, there were no common hardware platforms used to conduct performance or benchmarking experiments. One exception to this is in the ICAPS conferences, where the competition organizers ran the candidate solvers on a common computer in order to compare performance objectively.

Aside from the ICAPS competitions, it was found that some researchers did not take any runtime performance measurements at all, and of the researchers who did perform such measurements, usually the workstation that was available to that group was used. Commonly, researchers detailed the specifications of their computing systems. This approach seems to be the standard approach to date: each research group runs its algorithm on its own computing system, whatever it may happen to be.

With this approach, it becomes virtually impossible (without large amounts of reimplementation effort) to compare the runtime performance of alternative approaches introduced in two papers describing algorithms that are evaluated on different computing systems. There were no instances found where two research groups had the same computing hardware (in terms of processor type, CPU speed, RAM, etc.), even by coincidence.

5.2.4 Dimensionality of Rewards

An issue that further complicates the landscape is that of the dimensionality of the reward function. In Equation 3.1, $R(s_i, a)$ is the reward for selecting action a from state s_i. This is an example of a two-dimensional reward function, since the function is a mapping from $\mathcal{S} \times \mathcal{A}$ to a scalar reward value. However, as noted in [37] some works use a three-dimensional reward function $R(s_i, a, s_j)$, (a mapping from $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$) and some use a one-dimensional reward function $R(s_i)$, (a mapping from $\mathcal{S}$ only). The versions of Equation 3.1 that use three and one dimensional reward functions are shown in Equation 5.1 and Equation 5.2, respectively. These different representations can present logistical challenges when attempting to piece together various works.

$$V^n(s_i) = \max_{a \in \mathcal{A}} \{ \sum_{s_j \in \mathcal{S}} P(s_j | s_i, a) [R(s_i, a, s_j) + \beta V^{n-1}(s_j)] \} \tag{5.1}$$

$$V^n(s_i) = R(s_i) + \max_{a \in \mathcal{A}} \{ \beta \sum_{s_j \in \mathcal{S}} [P(s_j | s_i, a) V^{n-1}(s_j)] \} \tag{5.2}$$

For example, if an MDP in `.pomdp` format from [70] is imported into MATLAB using the `.pomdp` parser from [74], the reward function will exist in the MATLAB workspace as a three-dimensional rewards object. That representation is then incompatible with the MATLAB solver MDPSOLVE [33], which only accepts at most a two-dimensional reward function.

Conversion strategies to mitigate this are as follows. Increasing the dimensionality of the rewards is trivial, as the extra function input can simply be ignored.

Decreasing the dimensionality (as is required for the `.pomdp`/MDPSOLVE example above) requires some care. To reduce from a three-dimensional reward to a two-dimensional reward, Equation 5.1 is rewritten as Equation 5.3, and then the first summation is evaluated to arrive at the equivalent expression using a two-dimensional reward function in Equation 5.4. The resulting conversion formula is then shown in Equation 5.5.

$$\begin{aligned} V^n(s_i) = \max_{a \in \mathcal{A}} \{ \sum_{s_j \in \mathcal{S}} P(s_j | s_i, a) R(s_i, a, s_j) + \\ \beta \sum_{s_j \in \mathcal{S}} P(s_j | s_i, a) V^{n-1}(s_j) \} \end{aligned} \tag{5.3}$$

$$V^n(s_i) = \max_{a \in \mathcal{A}} \{ R(s_i, a) + \beta \sum_{s_j \in \mathcal{S}} P(s_j | s_i, a) V^{n-1}(s_j) \} \tag{5.4}$$

$$R(s_i, a) = \sum_{s_j \in \mathcal{S}} P(s_j | s_i, a) R(s_i, a, s_j) \tag{5.5}$$

This reduction can always be done without loss of information, and thus the constraint of at most a two-dimensional reward function is not a limitation for any solver. In spite of this, some solvers and parsers use three-dimensional rewards anyway.

There is no such formula to reduce from two-dimensional to one-dimensional rewards without loss of information. This can only be done in the special case where the content of the two-dimensional reward function is such that it is only a function of one of the arguments to begin with. In such a special case, the dimension can simply be collapsed along the dimension of the unused argument to convert to a one-dimensional reward function, typically $R(s)$.

Source	Number of MDPs	Format
SPUDD [47]	70	.spudd
pomdp-solve [70]	55	.pomdp
MDPSOLVE [33]	32	Constants in source code
Symbolic Perseus [74]	21	.spudd
ICAPS-04	20	PPDDL 1.0
ICAPS-16	15	RDDL
libpomdp [76]	15	.spudd and .pomdp
ZMDP [73]	9	.pomdp
AAPL [56]	8	.pomdp and .pomdpx
AIPS-02	8	PDDL 2.1
ICAPS-18	8	RDDL
AIPS-98	6	PDDL 1.0
AIPS-00	5	PDDL 1.0
ICAPS-06	10	PPDDL 1.0
ICAPS-11	11	RDDL
Noer13 [68]	2	.pomdp
AI-Toolbox [77]	4	Constants in source code
MDP Toolbox for MATLAB [72]	2	Constants in source code
C++ MDP Solver [75]	1	Constants in source code

Table 5.3: MDP datasets.

5.3 GEMBench

As motivated in Section 5.1, this section proposes the use of a common benchmarking platform called GEMBench. This section details the components of GEMBench, and justification for the decisions made in its design.

5.3.1 Selection Criteria

The selection of the hardware and operating system to target in the first version of GEMBench was made with the following considerations:

- Availability: the platform should be easily accessible to researchers, imposing minimal cost and logistical barriers.
- Repeatability: researchers should be able to reproduce published results from other researchers on their own platform instance, with their own experiments.
- Observability: researchers should be able to easily measure performance metrics that are relevant to ECS design, such as execution time, memory requirements, and power consumption.
- Ease of Use: A file system and robust networking stack is required to move MDP datasets and source code onto the platform with minimal effort.
- Development flexibility: the platform should be compatible with and contain rich support for toolchains of many different programming languages used in technical and scientific programming (e.g., C/C++, Python, Java, MATLAB, Go, Rust, Julia).

- GPU Support: It is reasonable to expect that GPUs will play a big role in the future advancements in MDP solvers, and thus the platform must have a programmable GPU.

- Documentation: The platform must be well documented in order to minimize the amount of initial time researchers need to spend to become productive.

- Long Term Support: The platform must have planned support for many years to come.

5.3.2 Hardware and Operating System

The common reference hardware of GEMBench, referred to as the GEMBench-compatible platform, is selected as the NVIDIA Tegra TX-1 Development Board. This platform is selected as one that satisfies the requirements summarized in Section 5.3.1. This is a Linux-based platform that contains a Quad ARM A57 CPU and an NVIDIA Maxwell GPU with 256 CUDA cores. The board contains 4GB of RAM and a 16GB eMMC storage. The board runs a Linux distribution known as Linux4Tegra (L4T), which is based on Ubuntu Linux. The software development kit provided with the board provides a well documented ecosystem.

The use of a self-contained single-board computer allows for computing hardware replication across research labs with minimal effort compared to setting up larger general purpose computing systems such as desktop computers. Additionally, the NVIDIA Jetson module is a small (50mm x 75mm) embedded computing module that can be designed into a small form factor ECS, outside of the development board. The Jetson module is well supported by a rich ecosystem of compatible peripherals including cameras, robotic platforms and drones.

The Linux-based OS provides a full-featured set of capabilities for software development and benchmarking, such as TCP/IP networking, USB, Wi-Fi, and HDMI video, to name a few. Linux is favored in this context over smaller embedded operating systems, such as Real-Time Operating Systems (RTOSs), due to its ease of use. Linux enables efficient use of common toolchains, and its file system and networking stack allow datasets to be copied onto the board easily. These two operations can be much more complicated on RTOS-based or smaller embedded OS systems. The lighter-weight OSs generally trade-off productivity in exchange for higher levels of optimization. In the design of GEMBench, ease of use is favored over optimization in this context. This is to provide a lower barrier to researchers getting up and running with the testbench.

5.3.3 Solvers

Part of the GEMBench Package [69] contains an open source release of an implementation of one of the GPU-accelerated solvers listed in Table 5.2: SPVI. This software package is intended to allow researchers who purchase the development board to easily reproduce the performance benchmarks of that solver. Additionally, the package contains guidance for how to contribute additional solvers that can be run on the platform. Specifically, the dependencies (and their versions) should be documented, along with compile and run instructions.

Ideally, researchers will be able to easily run any existing solvers, and then develop and implement new solvers on the platform, and easily produce benchmarking measurements that objectively compare multiple solvers on a robust collection of MDP datasets. The new performance claims can then be easily replicated across other research groups.

5.3.4 Datasets

A survey of datasets and their file formats used in MDP research was presented in Section 5.2. Due to the existing adoption of the `.pomdp` format across multiple research efforts, along with the large number of MDPs already available in that format, its continued use is encouraged here and it is selected for the primary benchmarking dataset. Also included is a curated set of `.pomdp` files as a packaged benchmarking dataset. Along with this set, the corresponding solutions to each of the MDPs is provided, which is something not previously found anywhere to date. The reference solutions were obtained using the pomdp-solve [70] and MDPSOLVE [33] solvers, due to their maturity over newer solver packages.

Also included in the GEMBench Package is an open source C/C++ example of how to ingest and parse MDP files in the `.pomdp` format. This example is intended to save researchers time in incorporating this file format into their solvers.

A secondary candidate for a benchmarking file format is the `.spudd` format, due to the large collection of MDP datasets available. As previously noted, this format is extremely difficult to work with due to its tree-shaped data structures and

Lisp syntax. There is interest in using the SPUDD MDPs for a secondary dataset, and this is left as a future research effort. There were two .spudd parsers found to date: one embedded in the SPUDD solver itself, and another provided by the Symbolic Perseus solver package [74] that can ingest .spudd files into MATLAB.

The current plan is to use these parsers to convert .spudd files into .pomdp files, although it is understood that some files will be very large in size after this conversion. Thus, only a subset will be converted. The benefit of this conversion is that any solver that can consume .pomdp files will also be able to solve the MDP datasets currently only available in .spudd format.

An open task is the parsing of .spudd files directly within a solver - any interested researcher is highly encouraged to contribute a utility that can ingest .spudd files and output them into more general-purpose format that does not depend so heavily on the concept of ADDs and Factorization that is central to the SPUDD solver.

Additionally, the survey in Section 3.4 and analyses in Section 4.8 strongly suggest that using sparse matrix representations would likely have a significant effect on MDP storage size and solver computation. Researchers are encouraged to propose a standard file format for sparse MDP representations and either convert existing .pomdp or .spudd MDPs into sparse formats, or contribute entirely new sparse MDPs to the collective datasets.

5.3.5 Measurements

Execution time can be measured on the board using one of two methods. One method is the use of software-based timestamps. Using these involves making calls to the Linux `time` API from user space, and finding the difference between successive calls to compute elapsed time.

Another method to measure time is the use of GPIO combined with an external oscilloscope. A GPIO line can be set from Linux user space at the start of a solver routine, and then cleared at the end. By measuring the resulting square wave voltage on this GPIO pin with an oscilloscope, a very precise timing measurement can be made.

Both CPU and GPU memory use can be measured using the NVIDIA CUDA API. The `cudaMemGetInfo()` function is well-documented and allows for objective measurement of memory consumption.

The power consumption of the entire board can be measured directly from Linux user space. The board contains a Texas Instruments INA3221 Current and Voltage monitor, and instantaneous values can be read directly from the device using a device driver provided by the NVIDIA board support package.

5.4 Experiments

To demonstrate the use of GEMBench, two MDP solvers were implemented on the NVIDIA Jetson TX-1 development board. The first implementation is a CPU-only implementation of Value Iteration, which is referred to here as VI. This was created directly from the equations detailed in [53]. This implementation represents a straightforward single-threaded implementation of the classical Value Iteration algorithm and was used purely as a baseline for comparison with the other implementation.

The second implementation choice was Sparse Parallel Value Iteration (SPVI), which was described in detail in Chapter 3. SPVI leverages the GPU for execution time acceleration. A key feature of SPVI is that it uses sparse representations and sparse matrix-vector arithmetic operations in the GPU.

In the experiments, the solver execution time was measured on the entire Cassandra dataset using Linux's native timing support. The results for the full dataset are shown in Figure 5.2. The curated Cassandra dataset has 55 MDPs on the testbench, and the results of solving all of these MDPs is shown in the figure. Each data point represents one invocation of the solver on a specific MDP.

The MDPs in Figure 5.2 are sorted by a scalar size metric $N_S^2 N_A$, where N_S and N_A are the number of states and actions in each MDP, respectively. $N_S^2 N_A$ is used to denote the size of the MDP because the largest data structures in an MDP are the state transition matrices, which contain a total of $N_S^2 N_A$ entries.

One can draw several conclusions from Figure 5.2. First, the GPU-based

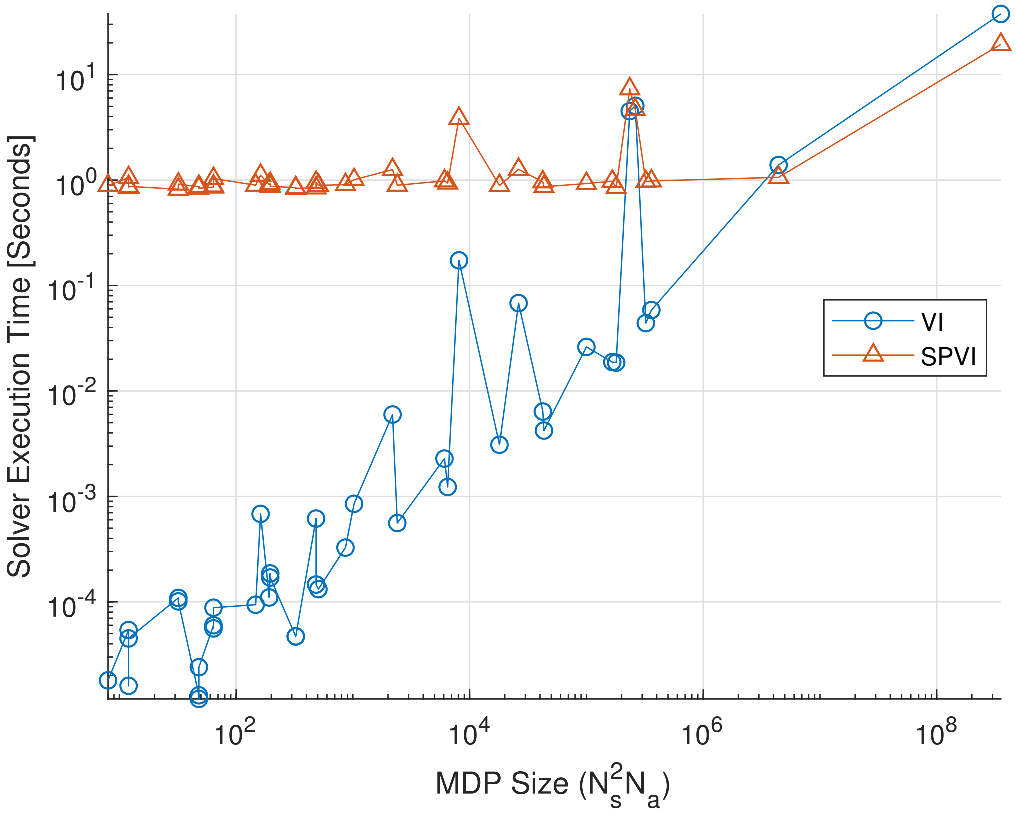

Figure 5.2: Solver execution time for Cassandra dataset.

implementation (SPVI) takes at least one second to solve any MDP, regardless of how small it is. This is due to the amount of time it takes to setup the GPU and initialize the CUDA `cuSparse` library used by SPVI. There does not appear to be any such setup time effect in the CPU-based VI algorithm.

Second, there is a crossover point around $N_S^2 N_A \approx 2e6$ beyond which the GPU-based solver is faster. To focus in on this aspect, the execution times for solving the four largest MDPs are listed in Table 5.4. For "baseball.pomdp" (the largest MDP in the dataset), SPVI (the GPU-based solver) is 48.5% faster than VI (the CPU-based solver), a considerable difference.

MDP Name	$N_S^2 N_A$	VI	SPVI
cit.pomdp	3.22e5	0.043	0.964
sunysb.pomdp	3.60e5	0.058	0.980
fourth.pomdp	4.42e6	1.390	1.061
baseball.pomdp	3.53e8	37.653	19.373

Table 5.4: Solver execution time (seconds).

Third, one can conclude that the Cassandra dataset is rich in small MDPs and lacking in large MDPs. Considerably more data is needed to explore GPU-based solver performance on larger MDPs. The GEMBench project aims to spur collaborative research in directions such as this, which support more insightful and comprehensive evaluation of alternative MDP implementation approaches.

5.5 Conclusion

This chapter has introduced GEMBench, a benchmarking tool for evaluating implementations of solvers for MDPs. The utility of common benchmarking environments has been demonstrated in many application areas. With the increasing relevance of MDPs in embedded systems, and the complex trade-offs involved in MDP solver deployment, GEMBench helps to bridge an important gap in design and implementation of MDP-equipped applications.

Along with the presentation of GEMBench, surveys of the landscape of MDP solvers, reference platforms and benchmarking were presented, with an emphasis on details that are relevant for experimenting with embedded implementations. GEMBench is designed for extensibility with additional file formats, datasets, and solvers, as well as retargetability to other processing platforms. Useful extensions for future work include continuing to add support for additional MDP file types and datasets, implementing more solvers from the literature, and exploring the trade-offs between using an MDP versus a POMDP model for a given ECS.

Additionally, the amount of system knowledge put into the MDP before deployment is an important consideration. An interesting area for future work relating to this is the exploration of making system assumptions offline versus learning those aspects of the system at runtime. The implementation aspects of these concepts with GPUs is likely to be an important research area in coming years.

Chapter 6

Conclusions and Future Work

In this thesis, a methodology for the design and implementation of runtime adaptation in embedded computing systems was presented. The methodology employs compact, system-level models based on Markov Decision Processes (MDPs) to generate control policies that optimize the required embedded computing tasks in terms of relevant, multidimensional design optimization metrics. Through simulations and implementations of case studies, the methodology was explored in several applications areas.

In Chapter 2, the methodology was applied to reconfigure a digital channelizer, and the results were shown to outperform the state-of-the-art in various metrics. One major aspect of the work in this chapter is that the channelization requests were treated as being generated externally, and the reconfigurable channelizer was only a subsystem tasked with responding to those requests. An interesting area for future research is in the incorporation of the runtime adaptation methodology to the upper layers as well, effectively creating an adaptive cognitive radio system. In that case, the channelization requests would be generated by the runtime adaptation framework in response to higher level application goals such as maintaining communications throughput in response to changing interference, as one example.

Surely, various new challenges would arise in that effort and those would be interesting areas for future research. Specifically, it can be anticipated that although the state representation used to encode the channelization requests (a bit vector, with one bit for each channel) worked well for the 8 channel case, the representation would likely be problematic for systems with larger numbers of channels. With 8 channels, the number of states required to represent the channelization request is

256, which is manageable in our design. However, for 16 or 24 channel systems, the size of the state space becomes a much more complex issue. It should be noted that this challenge is not remedied by the factorization techniques presented in Chapter 2, which were effective at keeping the state space small due to growth from other types of complexity. It is possible that a different encoding method can be used if some application-level context is incorporated. Progress in this direction may be possible with further analysis of a specific cognitive radio case study.

Additionally, the penalty costs of reconfiguration were modeled as being due to time spent reconfiguring the processing hardware, or channelization requests that were missed. If a full cognitive radio application were considered, there would likely be application level goals that could be used in the rewards, such as an end-to-end communications Quality of Service (QoS) objective, instead of or in addition to energy efficiency. In this form of multi-layered runtime adaptation, it is anticipated that some form of control and reconfiguration hierarchy would be beneficial.

In Chapter 3, an MDP solver was developed that solves certain MDPs faster than the state-of-the-art on embedded processors containing GPUs. The solver made use of the sparsity typically found in an MDP's transition matrices, and utilized efficient sparse linear algebra operations to achieve the fast runtime. This advancement was possible in part because of the relatively recent availability of general purpose programmable GPUs on embedded SoCs, compared with the multiple decades of MDP solver research on CPUs (without GPUs). Additionally, it was found that the Value Iteration (VI) algorithm is very well suited for acceleration with the parallel processing capabilities that GPUs can provide.

There are several directions for future work in this area. For example, an immediate question that arises is how well the acceleration scales for much larger MDPs. Certainly various optimization opportunities will arise when solving MDPs whose state spaces are much larger than the number of parallel threads that can be spawned on a GPU.

Aside from the intricacies that GPUs bring, there is much work left to do on periodic solving of time-varying MDPs. Specifically, in all cases in this thesis whenever an MDP was solved, it was solved without any prior knowledge of the solution. In other words, an MDP was solved to produce a policy; then the MDP was modified slightly (due to tracking of time-varying system or environmental factors); and then that second MDP was solved to produce a second policy, without any knowledge of anything produced by the first MDP solver invocation.

In the experimental observations, there was typically a high similarity or correlation between the solver's inputs from run to run. Correspondingly, it was also typical to see a level of similarity or correlation between the outputs. Given these trends, it is reasonable to hypothesize that some amount of the solver computations on subsequent runs are redundant in some way, and that there may be efficiency improvements in the solver algorithm that can be achieved if this redundancy could be understood and exploited — for example, if some artifact or intermediate calculations of the previous invocation are carried forward to the subsequent invocation.

The VI algorithm involves starting from a random choice of the Value Function and iterating towards a final solution. Typically, the starting point used was all zeros for consistency and to aid in debugging. However, it has been shown that the algorithm converges for any starting point. One approach that was tried was starting from the previous solution rather than from all zeros or a random point. In some preliminary experiments on this, the result was a faster solver time in some cases, and a slower solver time in other cases, and it was not clear why. Perhaps a starting point can be chosen more intelligently based on how the MDP has changed and some other aspects of the previous run's artifacts.

Another possible area for future work related to Chapter 3 is in the exploration of other solver algorithms altogether. This thesis has focused entirely on the VI algorithm, however there are other algorithms that have proved popular in the past, most notably Policy Iteration (PI) and Modified Policy Iteration (MPI). It remains to be seen whether these other algorithms can also benefit from being implemented in GPUs in the way that VI does.

Another aspect to consider is that these three algorithms (VI, PI, MPI) are known as exact solvers of MDPs. When viewed from an analytical standpoint, they converge on a provably optimal solution to an MDP. However, there also exists a class of MDP solvers known as approximate solvers, typically found in POMDP contexts. It remains to be seen whether the computational advantages of producing an approximate solution outweighs the disadvantages of producing a possibly suboptimal control solution in the context of embedded, runtime adaptation frameworks. The exploration of this trade-off on different applications and case studies could provide examples of important overall system improvements.

In Chapter 4, a runtime adaptation framework named Structured Learning was proposed and analyzed in the context of an LTE-M connected system. Simulations of control performance were compared to Q-Learning, a competing technique that has very recently been popularized in the research community. Additionally, the competing frameworks were implemented on a resource-constrained MCU and their memory and computational requirements were compared objectively.

There are various aspects of the work in Chapter 4 that generated new directions for future research. In the case study that was analyzed, the MCU contained a very simple application that generated packets in an idealized way. It would be interesting to dissect a relevant consumer product available today, and observe its traffic pattern. Then, attention could be paid as to how well the runtime adaptation framework could perform on that specific traffic pattern. If shortcomings are identified, then exploration of how to modify the runtime framework for that traffic pattern would be important. In addition to improving the performance of the given consumer application, this kind of study could provide useful insight into how methods proposed in this thesis can be further streamlined in application-specific ways.

Additionally, the work in Chapter 4 focused on adaptation of the LTE-M connection when the event of establishing the cellular modem's connection was time-varying. However, there are other parameters that would likely also be time-varying — for example, the transmit power, transmission latency and overall data rate. Modeling these additional parameters and re-defining the adaptation rewards in order to take them into account would be an interesting effort that is likely to yield new advances.

Another direction for future work stems from the fact that this thesis only compared Structured Learning to a basic form of Q-Learning. However, Q-Learning is a technique that exists in the larger area of Reinforcement Learning (RL), and that is a rapidly evolving area of research at this moment. It is anticipated that comparisons between Structured Learning and emerging advancements in RL in the context of embedded systems would prove insightful. As a relevant example, RL has found recent success [81] in beating the world champion of the board game Go by approximating the MDP Value Function using a deep neural network (DNN). While these techniques have traditionally been implemented on large computing systems with vast computing resources, recent advancements [82] have made it possible to implement DNNs on smaller and smaller computing systems, including extremely resource constrained MCUs. As a result, an interesting area for future research is in the design of runtime adaptation frameworks using DNN-based RL for highly resource constrained embedded computing systems, and how these techniques would compare to the Structured Learning techniques proposed in this thesis.

In Chapter 5, the GEMBench platform was created to aid in the development and exploration of techniques on embedded computing systems that use MDPs. The platform consists of software written throughout the work performed for this thesis research, targeted at a popular family of off-the-shelf NVIDIA embedded computing development boards.

The software has been released to the academic community in an open source format with a permissive license that allows it to be generally used by any researchers working in this area. The aim is to accelerate future developments by lowering the amount of time required to build critical software infrastructure, namely generating example MDPs, writing parsers to convert from one format to another, and implementing alternative solvers to compare policy outputs and computational requirements.

Several directions for future work remain in this area. Specifically, it would be good to add support for at least one more format for MDP files (`.spudd`). This file format is commonly found in other research efforts, and an interesting set of example MDPs could be immediately be added to the GEMBench platform.

Additionally, since the MDP state transition matrices are already located on the platform, it is possible to create a simulator that creates a model of a system and performs Monte Carlo analyses of the discrete time trajectories resulting from those state transition matrices. This would allow different policies to be evaluated in terms of the resulting closed loop dynamics. This capability currently does not exist on GEMBench. Instead, experiments requiring such capability were performed in MATLAB throughout the research involved in this thesis. The advantage of having this capability on GEMBench is that it would be easy to share the Markovian simulator with other researchers. It should be noted that this type of simulator could be used for exploring RL-based techniques, such as Q-Learning, as well.

MDPs are not always the most adequate modeling framework for every possible use of runtime adaptation. There are cases where they are highly effective, and cases where they are not. Various factors affecting the suitability of MDPs to specific applications were detailed in Section 3.4. For applications where the use of MDPs is advantageous, collaborative partnerships between researchers on common platforms like GEMBench can help to accelerate progress. As research and technology advance, applications will continue to arise that require solving bigger and more complicated runtime adaptation challenges. This will be aided partly by technology trends that result in faster and more efficient computing hardware. However, there is also an equally important category of continual advancement that must be made in algorithms and modeling techniques.

Bibliography

[1] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, June 2000.

[2] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, "Power management in energy harvesting sensor networks," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 4, 2007.

[3] C. Hsieh, F. Samie, M. S. Srouji, M. Wang, Z. Wang, and J. Henkel, "Hardware/software co-design for a wireless sensor network platform," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), International Conference on*, New Delhi, India, Oct 2014, pp. 1–10.

[4] Q. Liu et al., "Power-adaptive computing system design for solar-energy-powered embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 8, pp. 1402–1414, 2015.

[5] C. Moser, L. Thiele, D. Brunelli, and L. Benini, "Adaptive power management in energy harvesting systems," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2007, pp. 1–6.

[6] C. Moser, L. Thiele, D. Brunelli and L. Benini, "Adaptive power management for environmentally powered systems," *IEEE Transactions on Computers*, vol. 59, no. 4, pp. 478–491, April 2010.

[7] L. Esterle and B. Rinner, "An architecture for self -aware IoT applications," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2018, pp. 6588–6592.

[8] Nikil Dutt, Axel Jantsch, and Santanu Sarma, "Toward smart embedded systems: A self-aware system-on-chip perspective," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 2, pp. 22:1–22:27, Feb. 2016.

[9] P. R. Lewis, M. Platzner, B. Rinner, J. Torresen, and X. Yao, *Self-aware Computing Systems: An Engineering Approach*, Springer, 2016.

[10] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Dorling Kindersley, first edition, 1993.

[11] S. J. Darak, A. P. Vinod, R. Mahesh, and E. M-K. Lai, "A reconfigurable filter bank for uniform and non-uniform channelization in multi-standard wireless communication receivers," in *Proceedings of the International Conference on Telecommunications*, 2010, pp. 951–956.

[12] C. Xu, W. Liang, and H. Yu, "Green-energy-powered cognitive radio networks: Joint time and power allocation," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 1, pp. 13:1–13:18, Aug. 2017.

[13] A. Sapio, M. Wolf, and S. S. Bhattacharyya, "Compact modeling and management of reconfiguration in digital channelizer implementation," in *Proceedings of the IEEE Global Conference on Signal and Information Processing*, Washington, D.C., December 2016, pp. 595–599.

[14] A. Sapio, L. Li, J. Wu, M. Wolf, and S. S. Bhattacharyya, "Reconfigurable digital channelizer design using factored Markov decision processes," *Journal of Signal Processing Systems*, December 2017.

[15] C. Boutilier, R. Dearden, and M. Goldszmidt, "Exploiting structure in policy construction," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995, pp. 1104–1111.

[16] O. Sigaud and O. Buffet, Eds., *Markov Decision Processes in Artificial Intelligence*, Wiley, 2010.

[17] C. Lee, W. Chen, S. S. Bhattacharyya, and T. Lee, "Dynamic, data-driven spectrum management in cognitive small cell networks," in *8th International Conference on Signal Processing and Communication Systems (ICSPCS)*, Gold Coast, Australia, Dec 2014, pp. 15–17.

[18] J. Hu, Z. Zuo, Z. Huang, and Z. Dong, "Dynamic digital channelizer based on spectrum sensing," *PLOS One*, August 2015.

[19] D. Zhou, "A review of polyphase filter banks and their application," Tech. Rep. AFRL-IF-RS-TR-2006-277, Air Force Research Laboratory, Rome, NY USA, September 2006.

[20] F. Harris, C. Dick, X. Chen, and E. Venosa, "Wideband 160-channel polyphase filter bank cable TV channeliser," *IET Signal Processing*, vol. 5, no. 4, pp. 325–332, 2011.

[21] F. Harris, E. Venosa, X. Chen, C. Dick, and B. Adams, "A novel and efficient multi-resolution channelizer for software defined radio," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2013, pp. 2649–2653.

[22] S. Dhabu, Smitha K. G., and A. P. Vinod, "A low complexity reconfigurable channel filter based on decimation, interpolation and frequency response masking," in *IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP)*, Vancouver, BC, Canada, May 2013, pp. 5583–5587.

[23] A. Edison and T. G. James, "Reconfigurable perfect reconstruction filter bank channelizer for software defined radio," in *IEEE India Conference (INDICON)*, Kochi, India, Dec 2012, pp. 1138–1141.

[24] W. A. Abu-Al-Saud and G. L. Stuber, "Efficient wideband channelizer for software radio systems using modulated PR filterbanks," *IEEE Transactions on Signal Processing*, vol. 52, no. 10, pp. 2807–2820, 2004.

[25] Z. Chang, A. P. Vinod, and P. K. Meher, "Reconfigurable architectures for low complexity software radio channelizers using hybrid filter banks," in *Proceedings of the IEEE Singapore International Conference on Communication Systems*, 2006, pp. 1–5.

[26] S. J. Darak, S. K. P. Gopi, V. A. Prasad, and E. Lai, "Low-complexity reconfigurable fast filter bank for multi-standard wireless receivers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 5, pp. 1202–1206, 2014.

[27] P. K. Devi and R. S Bhuvaneswaran, "Flexible reconfigurable architecture for SDR receiver," in *IEEE Malaysia International Conference on Communications (MICC)*, Kuala Lumpur, Nov 2013, pp. 265–270.

[28] Y. Wei, X. Wang, F. Guo, G. Hogan, and M. Collier, "Energy saving local control policy for green reconfigurable routers," in *IEEE International Conference on Communications*, 2015, pp. 221–225.

[29] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*, Prentice Hall, second edition, 1999.

[30] M. Xu, H. Li, and X. Gan, "Energy efficient sequential sensing for wideband multi-channel cognitive network," in *IEEE International Conference on Communications*, 2011, pp. 1–5.

[31] B. Farzad, "Variable bandwidth polyphase filter banks," M.S. thesis, San Diego State University, 2014.

[32] B. E. Bjornson, E. A. Jorswieck, M. Debbah, and B. Ottersten, "Multiobjective signal processing optimization: The way to balance conflicting metrics in 5G systems," *IEEE Signal Processing Magazine*, vol. 31, no. 6, pp. 14–23, 2014.

[33] P. L. Fackler, "MDPSOLVE a MATLAB toolbox for solving Markov decision problems with dynamic programming — user's guide," Tech. Rep., North Carolina State University, January 2011.

[34] L. Benini, A. Bogliolo, A. Paleologo, and G. De Micheli, "Policy optimization for dynamic power management," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 813–833, June 1999.

[35] S. Russell and P. Norvig, *Artificial Intelligence, A Modern Approach*, Pearson, third edition, 2009.

[36] C. M. Grinstead and J. L. Snell, *Introduction to Probability*, American Mathematical Society, second edition, 1991.

[37] Andrew Y. Ng, *Shaping and Policy Search in Reinforcement Learning*, Ph.D. thesis, University of California, Berkeley, 2003.

[38] A. Sapio, S. Bhattacharyya, and M. Wolf, "Efficient solving of Markov decision processes on GPUs using parallelized sparse matrices," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Oct 2018.

[39] A. Munir and A. Gordon-Ross, "An MDP-based application oriented optimal policy for wireless sensor networks," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2009, pp. 183–192.

[40] A. Singh, A. Prakash, K. Basireddy, G. Merrett, and B. Al-Hashimi, "Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on cpu-gpu mpsocs," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 147:1–147:22, Sept. 2017.

[41] S. Wang, G. Zhong, and T. Mitra, "CGPredict: Embedded GPU performance estimation from single-threaded applications," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 146:1–146:22, Sept. 2017.

[42] S. Ruiz and B. Hernandez, "A parallel solver for Markov decision process in crows simulations," in *Artificial Intelligence (MICAI), Fourteenth Mexican International Conference on*, Mexico, Octoober 2015, pp. 107–116.

[43] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, pp. 30:1–30:49, Jan. 2017.

[44] Y. Debizet, G. Lallement, F. Abouzeid, P. Roche, and J. Autran, "Q-learning-based adaptive power management for IoT system-on-chips with embedded power states," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[45] E. Jung, F. Maker, T. L. Cheung, X. Liu, and V. Akella, "Markov decision process (MDP) framework for software power optimization using call profiles on mobile phones," *Journal of Design Automation for Embedded Systems*, vol. 14, no. 2, pp. 131–159, 2010.

[46] Z. N. Sunberg, M. J. Kochenderfer, and M. Pavone, "Optimized and trusted collision avoidance for unmanned aerial vehicles using approximate dynamic programming," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 1455–1461.

[47] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, "SPUDD: stochastic planning using decision diagrams," in *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1999, pp. 279–288.

[48] A. Jonsson and A. Barto, "Causal graph based decomposition of factored MDPs," *Journal of Machine Learning Research*, vol. 7, pp. 2259–2301, 2006.

[49] T. Dean and S. Lin, "Decomposition techniques for planning in stochastic domains," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, 1995, IJCAI'95, pp. 1121–1127.

[50] A. Somani, N. Ye, D. Hsu, and W. Sun Lee, "DESPOT: online POMDP planning with regularization," *Advances in Neural Information Processing Systems*, vol. 58, 01 2013.

[51] Anton Wijs, Joost-Pieter Katoen, and Dragan Bošnački, "Efficient GPU algorithms for parallel decomposition of graphs into strongly connected and maximal end components," *Formal Methods in System Design*, vol. 48, no. 3, pp. 274–300, Jun 2016.

[52] R. Bellman and E. Lee, "History and development of dynamic programming," *IEEE Control Systems Magazine*, vol. 4, no. 4, pp. 24–28, November 1984.

[53] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, John Wiley and Sons, Inc., first edition, 2005.

[54] S. Shresthamali, M. Kondo, and H. Nakamura, "Adaptive power management in solar energy harvesting sensor node using reinforcement learning," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 181:1–181:21, Sept. 2017.

[55] P. Wägemann, T. Distler, H. Janker, P. Raffeck, V. Sieh, and W. SchröDer-Preikschat, "Operating energy-neutral real-time systems," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 1, pp. 11:1–11:25, Aug. 2017.

[56] O. Brock, J. Trinkle, and F. Ramos, *SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces*, MIT Press, 2009.

[57] T. Smith and R. Simmons, "Point-based POMDP algorithms: Improved analysis and implementation," in *Proc. of the Conference on Uncertainty in Artificial Intelligence*, July 2005.

[58] R. Sutton and A. Barto, *Reinforcement Learning: an Introduction*, MIT Press, first edition, 1998.

[59] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for Q-Learning in random stochastic mazes," in *Proceedings of the IEEE Symposium Series on Computational Intelligence (SSCI)*, Dec 2016, pp. 1–8.

[60] R. C. Hsu, C. T. Liu, and H. L. Wang, "A reinforcement learning-based ToD provisioning dynamic power management for sustainable operation of energy harvesting wireless sensor node," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 2, pp. 181–191, June 2014.

[61] W. Liu, Y. Tan, and Q. Qiu, "Enhanced q-learning algorithm for dynamic power management with performance constraint," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2010, pp. 602–605.

[62] S. Yue, D. Zhu, Y. Wang, and M. Pedram, "Reinforcement learning based dynamic power management with a hybrid power supply," in *Proceedings of the IEEE international Conference on Computer Design (ICCD)*, Sept 2012, pp. 81–86.

[63] J. Modayil, A. White, P. M. Pilarski, and R. S. Sutton, "Acquiring a broad range of empirical knowledge in real time by temporal-difference learning," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct 2012, pp. 1903–1910.

[64] Theodore J. Perkins and Andrew G. Barto, "Lyapunov design for safe reinforcement learning," *J. Mach. Learn. Res.*, vol. 3, pp. 803–832, Mar. 2003.

[65] Yinlam Chow, Ofir Nachum, Edgar Duenez-Guzman, and Mohammad Ghavamzadeh, "A Lyapunov-based approach to safe reinforcement learning," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, USA, 2018, NIPS'18, pp. 8103–8112, Curran Associates Inc.

[66] S. Dawaliby, A. Bradai, and Y. Pousset, "In depth performance evaluation of LTE-M for M2M communications," in *Proceedings of the IEEE 12th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, Oct 2016, pp. 1–8.

[67] A. Sapio, R. Tatiefo, S. Bhattacharyya, and M. Wolf, "GEMBench: a platform for collaborative development of GPU accelerated embedded Markov decision systems," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, Jul 2019.

[68] D. Noer, "Parallelization of the value-iteration algorithm for partially observable markov decision processes," M.S. thesis, Technical University of Denmark, 2013.

[69] "GEMBench Package," 2019, `https://ece.umd.edu/DSPCAD/projects/csm/packages/gembench.tar.gz`, Visited on March 14, 2019.

[70] A. R. Cassandra, *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*, Ph.D. thesis, Brown University, 1998.

[71] Hyeong Seop Sim, Kee-Eung Kim, Jin Hyung Kim, Du-Seong Chang, and Myoung-Wan Koo, "Symbolic heuristic search value iteration for factored POMDPs," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2008.

[72] Kevin Murphy, "Markov decision process toolbox for MATLAB," `https://www.cs.ubc.ca/~murphyk/Software/MDP/mdp.html`, Accessed: 2019-01-05.

[73] T. Smith, *Probabilistic Planning for Robotic Exploration*, Ph.D. thesis, Carnegie Mellon University, 2007.

[74] P. Poupart, *Exploiting Structure to Efficiently Solve Large Scale Partially Observable Markov Decision Processes*, Ph.D. thesis, University of Toronto, 2005.

[75] P. Elod, "Vision-based quadcopter navigation for following indoor corridors and outdoor railways," M.S. thesis, Tech. Univ. of Cluj-Napoca, 2014.

[76] Diego Maniloff, "libpomdp," `https://www.cs.uic.edu/~dmanilof/code.html`, Accessed: 2019-01-05.

[77] E. Svalorzen, "AI-Toolbox," `https://github.com/Svalorzen/AI-Toolbox`, Accessed: 2019-01-05.

[78] Drew McDermott, *PDDL — The Planning Domain Definition Language*, Yale University, 1998.

[79] Hakan L. S. Younes and Michael L. Littman, "PPDDL1.0: the language for the probabilistic part of ipc-4," Tech. Rep.

[80] Scott Sanner, "Relational dynamic influence diagram language (RDDL): Language description," 2010.

[81] D. Silver et al., "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354, Oct 2017.

[82] Liangzhen Lai and Naveen Suda, "Enabling deep learning at the IoT edge," in *Proceedings of the International Conference on Computer-Aided Design*, 2018, ICCAD '18.