



AFRL-RI-RS-TR-2021-165

PROTEUS: CONTROLLING RESOURCE ADAPTIVE EMBEDDED SYSTEMS

RICE UNIVERSITY

SEPTEMBER 2021

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2021-165 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

STEVEN L. DRAGER
Work Unit Manager

/ S /

GREGORY J. HADYNSKI
Assistant Technical Advisor,
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) SEPTEMBER 2021		2. REPORT TYPE FINAL TECHNICAL REPORT		3. DATES COVERED (From - To) NOV 2015 – MAR 2021	
4. TITLE AND SUBTITLE PROTEUS: CONTROLLING RESOURCE ADAPTIVE EMBEDDED SYSTEMS				5a. CONTRACT NUMBER FA8750-16-2-0004	
				5b. GRANT NUMBER N/A	
				5c. PROGRAM ELEMENT NUMBER 61101E	
6. AUTHOR(S) Krishna Palem (Rice University), Robert Cartwright (Rice University), Arvind (Massachusetts Institute of Technology), Hank Hoffmann (University of Chicago), Don Fussell (University of Texas at Austin)				5d. PROJECT NUMBER BRAS	
				5e. TASK NUMBER RI	
				5f. WORK UNIT NUMBER CE	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rice University 6100 Main St Houston TX 77005				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA DARPA 525 Brooks Road 675 N. Randolph St Rome NY 13441-4505 Arlington VA 22203-2114				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI	
				11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2021-165	
12. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT The goal of the Proteus project was to develop a software framework and embedded hardware platform for constructing streaming applications that dynamically adapt to changes in the operating environment and ecosystem. The project developed a programming language, FAST, and runtime for constructing adaptive systems that are able to meet a specified mission intent, such as meeting constraints on measures (e.g. latency or energy) while optimizing an objective function (over any measures), and doing so in spite of perturbations in the operating environment, or test parameters such as mission length. The technologies were empirically validated, demonstrated in the context of challenge problems, and evaluated by an independent team.					
15. SUBJECT TERMS Adaptive Software Systems, Dynamic Adaptation of Embedded Applications, Software Discovery and Analysis, Mission Intent					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 136	19a. NAME OF RESPONSIBLE PERSON STEVEN L. DRAGER
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) N/A

Table of Contents

1	Summary	1
2	Introduction	2
3	Methods, Assumptions, and Procedures	4
3.1	Technical Area: Platform	4
3.1.1	Task 1.1: Platform Emulation Supporting Proteus Development	4
3.1.1.1	PEMU	4
3.1.1.2	PEMU API Layer	8
3.1.2	Task 1.2: Emulation Platform and Cost Modeling	8
3.1.2.1	Preliminary Investigation	9
3.1.2.2	Revised Methodology	10
3.1.2.3	Enhancements	12
3.1.2.4	Conclusion	12
3.1.3	Task 1.3: Machine-Learning Drone Design and Validation	13
3.1.3.1	Low-level Software (Firmware and Operating System)	13
3.1.3.2	Swift Compiler	13
3.1.3.3	Other Libraries and Dependencies	14
3.1.4	Task 1.4: Interfacing Machine-Learning Drone Design with Emulator	14
3.1.5	Task 1.5: Machine-Learning Drone Laboratory Validation Using TA3 and TA2 Outcomes	15
3.1.5.1	Platform Improvements	15
3.1.5.2	Porting FAST and the Challenge Problems	16
3.1.5.3	Platform-Specific Interfaces	16
3.2	Technical Area: Intent-Aware Discovery Through Machine Learning	17
3.2.1	Tasks 2.1 & 3.2: Interface between TA2 and TA3 Innovations	17
3.2.2	Task 2.2: ϵ Approximate Control Theoretic Framework and Assurance Monitor Design	19
3.2.2.1	ϵ Approximate Control Theoretic Framework	19
3.2.2.2	Assurance Monitor Design	22
3.2.3	Task 2.3 & 2.5: Resilient and Adaptive Memory Models	22
3.2.3.1	Why Programs Break Down on New Embedded Platforms	22
3.2.3.2	Programming in the C++11 Language Model	24
3.2.3.3	WMM: A New Resilient Memory Model for Processors	25

3.2.4 Task 2.4: Integrated Validation of Control Theoretic Framework with Machine-Learning Discovery and Assurance Monitoring	28
3.2.4.1 Combining Learning and Control	28
3.2.4.2 Improving Learning for Computer Systems Discovery	30
3.2.5 Task 2.6: Assurance Monitor to Trigger Transitions from Online to Offline Learning Methods	31
3.3 Technical Area: Novel Control-Theoretic Analytics Supported by Machine Learning	33
3.3.1 Task 3.1: Offline Machine-Learning Models and Model Coherence with Online Learning	33
3.3.1.1 Problem Formulation	33
3.3.1.2 Proactive Control for Approximate Programs	35
3.3.2 Task 3.3: Definition of the FAST Language and Implementation of Selected Features	37
3.3.2.1 Intent-Driven Programming Model	37
3.3.2.2 FAST Language Definition	39
3.3.2.3 METAFast Language Extension	39
3.3.2.4 Challenge Problems	39
3.3.2.5 Results and Accomplishments	40
3.3.2.6 Hierarchical Intents and Control through Proteus	43
3.3.3 Task 3.4: Online Learning Algorithms Transferred to Analytics Team	43
3.3.3.1 Overview	43
3.3.3.2 Accounting for Learned Models	48
3.3.3.3 Offline Learning	49
3.3.3.4 Kalman filtering for Proteus	49
3.3.3.5 Updating Machine Learning Models Online	50
3.3.4 Task 3.5: Implementation of Relevant FAST Language Features and Testing Interaction with the Machine-Learning System	51
3.3.4.1 Integration with Machine-Learning System	52
3.3.5 Task 3.6: Offline and Online Machine-Learning Algorithms, Model Refinement, and Intent-Aware Programming for Demo Preparation	52
3.3.6 Effectiveness of Online Machine Learning Algorithms	54
3.3.7 Adaptation as a First-class Programming Construct	56
4 Results and Discussion	59
4.1 Evaluation	59
4.1.1 Tasks 4.1, 4.2, 4.3, 4.4, 4.5 and 4.9: Challenge Problems for Phases I, II and III	59
4.1.1.1 SAR: Synthetic Aperture Radar	59
4.1.1.2 x264: Video encoding	60
4.1.1.3 CaPSuLe: Image matching	60
4.1.1.4 flightTestScenario7: Radio Bandwidth Scheduling	60
4.1.2 Task 4.6: Testing of Proteus with Integrated Memory Models on Hardware Drone	61
4.1.3 Task 4.7: Integrated System Validation Through Metrics	64

4.1.3.1 Mean Combined Relative Constraint Error (MCRCE)	64
4.1.3.2 Plasticity	65
4.2 Porting and Demonstration	67
4.2.1 Task 4.8: Porting and Platform Validation	67
4.2.1.1 Porting	67
4.2.1.2 Platform Validation	67
4.2.2 Task 4.10: Demonstration	68
4.2.2.1 Phase I Platform Workshop Demo	68
4.2.2.2 Phase II Platform Workshop Demo	68
4.2.2.3 Phase III Platform Workshop Demo	68
5 Conclusions	72
5.1 Lessons Learned	72
5.1.1 Solving Constraint Optimization Problems	72
5.1.2 Handle Input-Dependence through Online Learning	72
5.1.3 Model Size	73
5.1.4 Modularity Through High-Level Interfaces	73
5.1.5 Treat Application and Hardware as Equals	73
5.1.6 Expose Adaptation as a First-Class Object	73
5.2 Key Contributions and Achievements	74
5.2.1 Key Contributions in Combining Learning and Control	74
5.2.2 Achievements in ϵ Approximate Control Theory	74
5.2.3 Key Contributions of the WMM work	74
Appendix: Proteus Test Plan	76
Appendix: Challenge Problem SAR	94
Appendix: Challenge Problem x264	103
Appendix: Challenge Problem CaPSuLe	110
Appendix: Challenge Problem flightTestScenario7	114

Table of Figures

1 Proteus System Overview	2
2 PEMU.	4
3 PEMU ER.	6
4 X264 App Running Under PEMU	10
5 A Client App Interacting With x264 and PEMU	11
6 Performance Hash Table	18
7 Reader Scrambling Input	21
8 (a) A Streaming Model For Semidefinite Programs (b) Injecting Inexactness Into SDP-Based Approximation Algorithms for CSPs	22
9 Message Passing in C++11 and Different ISAs	25
10 WMM Abstract Machine	26
11 Mapping C++11 to WMM	26
12 Operations of the WMM Abstract Machine	27
13 Overview of the Capri Control System	36
14 An Example of METAFast intent files.	40
15 The results of running two RDFT functions with different intent goal ($MSE \leq 100$ (red) and $MSE \leq 150$ (blue) controlled by METAFast.	41
16 The results of running two RDFT functions with different intent goal ($MSE \leq 100$ (red) and $MSE \leq 200$ (blue) controlled by METAFast.	42
17 The results of running one RDFT functions with intent goal $MSE \leq 100$ (red) and one RLS with intent goal $RE \leq 0.3$ (blue) controlled by METAFast.	44
18 The results of running one invocation of RDFT functions with intent goal $MSE \leq$ 100 (red) and one invocation of RLS with intent goal $RE \leq 0.3$ (blue) controlled by METAFast.	45
19 The results of running one invocation of RDFT functions with intent goal $MSE \leq$ 100 (red) and its caller, an invocation of LPF with intent goal $MSE \leq 200$ (blue), both controlled by METAFast.	46
20 Schematic Diagram for Combining Learning and Control	47
21 Online Communication of Learning and Control	47
22 Effectiveness of the Online Machine Learning Module	55
23 Types of Adaptation	56
24 H.264 Pipeline	60
25 Performance Improvement by Removing Load-to-Store Ordering	62
26 Performance Improvement by Removing Same-Address Load-Load Ordering	63
27 Proteus Phase III Evaluation Report	69
28 The Graphical User Interface of the Phase I Demonstrator Application	70

29 The Graphical User Interface of the Phase II Demonstrator Application	70
30 SwRI's Radio Queue Simulator Showing Bandwidths Selected by Proteus	71
31 Derivation of achievable mission length	81
32 Generating achievable goals for a constraint with two measures m_1 and m_2 by randomizing schedules	89
33 Behavior of the Phase III oracle	98
34 Phase III verdict expression	100

Table of Tables

1 libpemu_client API	7
2 PEMU Test Adapter Interface	8
3 PEMU Public Interace	9
4 ReST Interface of the Online Machine Learning Module	53
5 Processor Parameters	62
6 Performance of WMM vs. TSO on radar and x264	63
7 Examples of Platforms to Which Proteus was Ported	67
8 Common Test Parameters	77
9 Definition of D_m	79
10 Optimization Problem Per Challenge Problem	79
11 Test Cases for the missionIntent Parameter	81
12 Goal and Maximal Post-Perturbation Mission Length Per Challenge Problem	83
13 Test Cases for the missionLength Parameter	83
14 sceneImportance Test Case Performance Goals Per Challenge Problem	83
15 Test Cases for the sceneImportance Parameter	84
16 Bounds on Achievable Performance Goals When utilizedCores is 2	84
17 Test Case for availableCores Parameter	84
18 Bounds on Achievable Performance Goals Where utilizedCoreFrequency is 600	85
19 Test Case for the availableCoreFrequency Parameter	85
20 Common Test Parameters	87
21 Window Sizes for CP2 and CP3	89
22 Definition of D_m	89
23 Optimization Problem per Challenge Problem	90
24 Test Cases for the missionIntent Parameter	91
25 Bounds on Achievable Performance Goals When utilizedCores is 2	92
26 Test Case for the availableCores Parameter	92
27 Bounds on Performance Goals When utilizedCoreFrequency is 600	92
28 Test Case for the availableCoreFrequency Parameter	93
29 Challenge Problem SAR Test Parameters	96
30 Challenge Problem SAR Knobs	97
31 Challenge Problem SAR Verdict Expression Parameters	99
32 Challenge Problem x264 Test Parameters	104
33 Challenge Problem x264 Knobs	105
34 Challenge Problem x264 Verdict Expression Parameters	106
35 Challenge Problem CaPSuLe Test Parameters	111
36 Challenge Problem CaPSuLe Knobs	112

37 Challenge Problem CaPSuLe Verdict Expression Parameters	113
38 Challenge Problem flightTestScenario7 Test Parameters	115
39 Challenge Problem flightTestScenario7 Knobs	115
40 Challenge Problem flightTestScenario7 Verdict Expression Parameters	116

1 Summary

The principal goal of the Proteus project is to develop a software framework and embedded hardware platform for constructing streaming applications that dynamically adapt to changes in the operating environment. This report summarizes the activity of all the tasks in the project, with sections for each task, as well as a summary of the evaluation that Lincoln Laboratory orchestrated toward the end of the phase.

The project developed a programming language and runtime (Section 3.3.4) for constructing adaptive systems that are able to meet a specified mission *intent*. This intent entails meeting constraints on measures (*e.g.* latency or energy) while optimizing an objective function (over any measures), and do so in spite of perturbations in the operating environment, or test parameters such as mission length. To support constructing and evaluating such applications, the project developed a hardware platform emulator, which relies on models of performance and energy usage on the target hardware platform. The Proteus system relies on machine learning: offline to obtain models used by an online feedback controller to select system configurations that will satisfy the intent; online to update stale models (Sections 3.3.3 and 3.3.5).

Proteus comprises a hardware emulation component (Section 3.1.2) capable of emulating the Proteus target platform (Section 3.1.3) during the evaluation of challenge problems in the Amazon test environment, as well as on development machines. The challenge problems that were used in the evaluation (Section 4.1.1) were ported to the target platform to enable demonstration as well as for gathering of data needed for emulation (Section 3.1.5).

These challenge problems were based on software packages that make use of parallel processing facilities of the target hardware platform. Such applications are known to be difficult to port to a new hardware platform without introducing bugs. The project developed a memory model that is resilient to such platform migration (Sections 3.2.3 and 4.1.2).

A framework for assurance monitoring was developed and validated through a prototype implementation, using an application from the autonomous navigation domain (Section 3.2.5).

Innovations developed throughout the project are brought together by the programming language FAST (Section 3.3.4), an instance of the novel *intent-driven programming* model, where application execution is supported by a runtime that adapts the behavior of the application, and the system it is running on, to meet a dynamically changeable intent specified in the language.

The Proteus system underwent thorough empirical validation in internal tests that mirrored the evaluation that was conducted by Lincoln Laboratory (Section 3.2.4).

2 Introduction

The Proteus project focused on developing innovative technologies supporting dynamic adaptation of embedded applications to changes in their resources or operating environments while maximizing the fidelity of their output. Over both long-term and short-term periods, should resources become less available, an application's fidelity could decrease, while it could increase if provided with an increase in resources. The Proteus system embodies a novel synergy between machine learning-based discovery resulting in models of an application's behavioral state space, and control system-based analytics that manage the application's adaptation to its environment using discovered models. Our resulting embedded system PROTEUS supports development of adaptive applications through discovery of programmer intent for specifying the desired quality and performance trade-offs used by applications for adaptation. To this end, our software discovery and analysis components have a front end, FAST, embodying programming language extensions to Swift designed to capture user intent and specify adaptive behaviors, as well as supporting compilation to lightweight run time components for adaptive control.

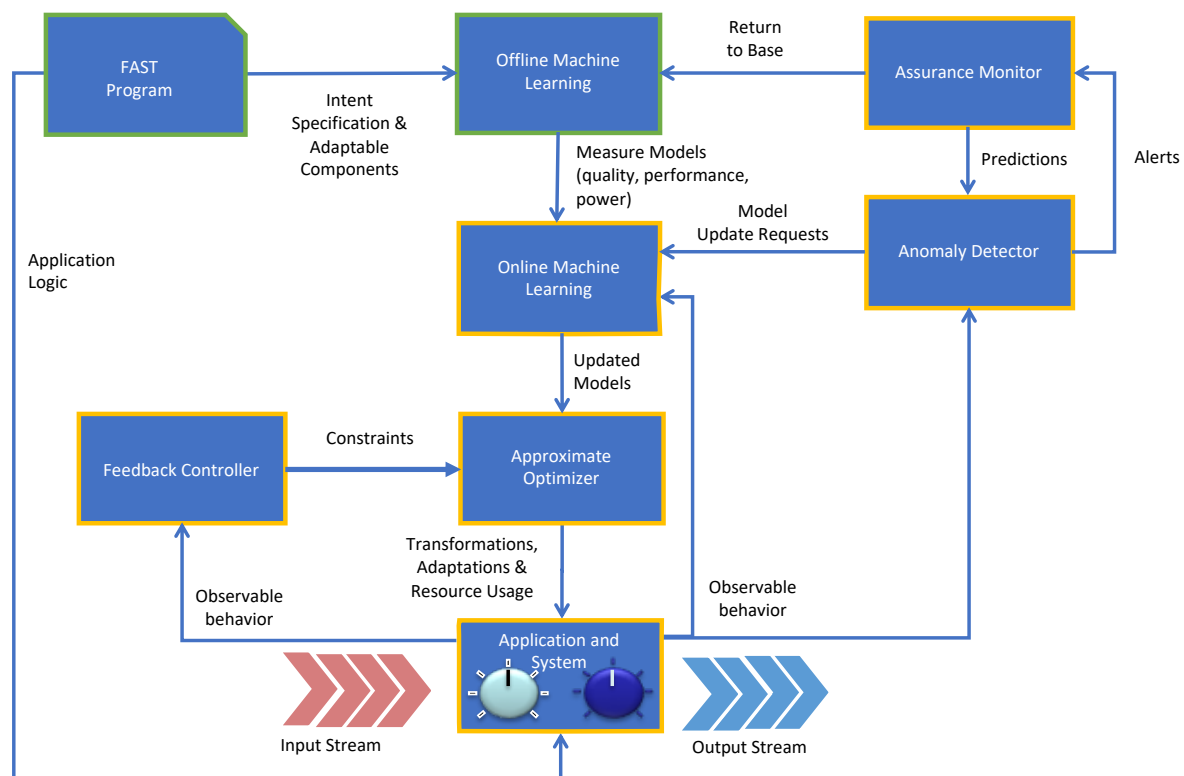


Figure 1: Proteus System Overview

Figure 1 shows the main components of the Proteus system and their interfaces. A Proteus application is a program written in the FAST language (Section 3.3.2), and consists of the basic application logic, some instrumentation code to enable adaptation, and a specification of intent. Before execution, offline machine learning (Section 3.3.4) is used to obtain a model that maps the different configurations exposed by the application and system, to different measures, such as performance, energy consumption and quality. This model, which can be updated online (Section 3.3.5), is used to determine how the application and system should be configured. The approximate optimizer does this using a feedback controller to make Proteus robust to systematic bias in the model, for example due to changes in the ambient workload on the system. During execution, the need for online learning is triggered by an anomaly detector, and an assurance monitor (Section 3.2.5) predicts a future violation of a safety property, to trigger a return to base for more offline learning. The evaluation (Sections 4.1.2, 4.1.3, and 4.2.1) is supported by a Representational State Transfer (REST) interface through which a test harness can initialize and perturb the system, for example to disable cores or change the intent specification, and receive messages containing comprehensive status information, as well as a final verdict computed by the application.

The rest of this report breaks down the work performed in the project along three technical areas: Platform (3.1), Intent-Aware Discovery Through Machine Learning (3.2) and Novel Control-Theoretic Analytics Supported by Machine Learning 3.3. The final part (4) covers evaluation tasks, which span multiple technical areas.

3 Methods, Assumptions, and Procedures

3.1 Technical Area: Platform

3.1.1 Task 1.1: Platform Emulation Supporting Proteus Development

The primary artifact under development in the Proteus project is a platform for executing streaming applications that dynamically adapt to changes in the operating environment. To facilitate software development and testing, the project team has developed an emulator for the platform that can run these applications including the simulation of dynamic changes in the environment. This emulator, called PEMU, simulates changes in the environment as fluctuations in the sensor readings and hardware resources. The supporting database enables the emulator to model some differences in hardware platforms including the number of available CPU cores, CPU clock rates, and the energy usage of each machine operation.

Two instances of PEMU were developed in the project. The first instance was written in C for compatibility with the POET [24] control system, and is documented in subsection 3.1.1.1. The second instance was written in Swift, integrated into the FAST runtime, and is documented in Section 3.8 of [65].

3.1.1.1 PEMU

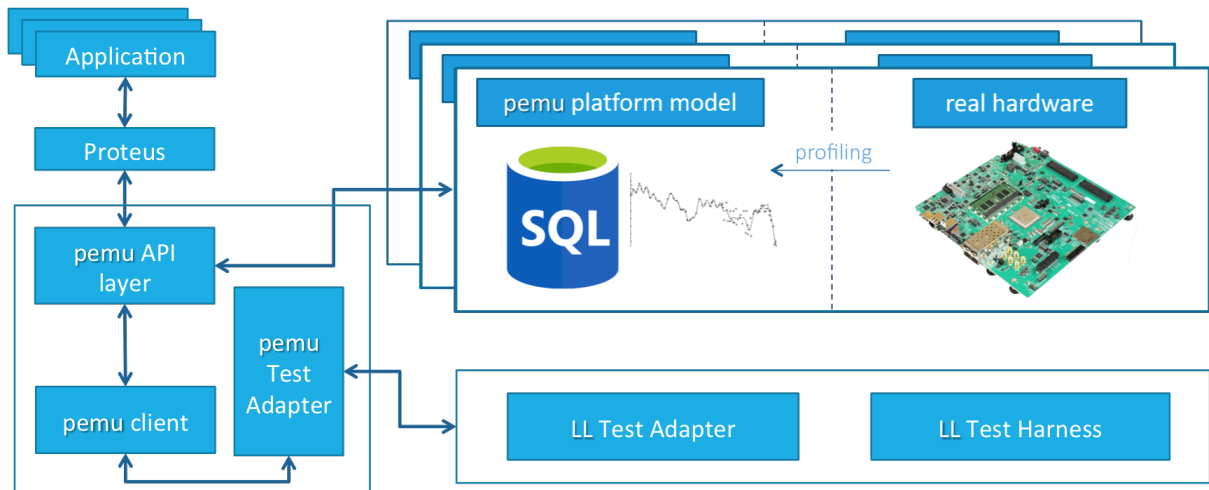


Figure 2: PEMU.

Figure 2 depicts the structure of PEMU, which consists of:

1. a platform model incorporating an SQL database containing profiled data from running various applications on real hardware;

2. a client module, which provides asynchronous communication with external clients via a TCP connection;
3. a Test Adapter module to interact with the Test Harness (TH) via RESTful APIs; and
4. an API layer, which maintains the emulator state and provides public APIs to access the emulator state and to set up and tear down: the database connection, the client TCP connection, and the connection with the test harness server.

The rest of this section describes each of these four components.

PEMU Database

As an application runs on the hardware platform, an embedded control system monitors its behavior and dynamically adjusts the system configuration parameters and application parameters so that the application can meet unexpected new requirements and adapt to unexpected environment changes. To emulate this behavior, PEMU provides platform-specific measurements such as the time and the energy it takes an application to process each input unit of the input streams.

To this end, PEMU relies on a database constructed by testing the application on the targeted real hardware platforms. The database records profile data for various system configurations and various application parameters. Ideally, the profiling should be done on the cross product of the system configuration parameters and the application parameters and a non trivial set of input streams. However this construction is impractical due to the large space of parameters. To work around this issue, we first limit the input stream to one for each of the two challenge applications. We then profile two sets of data: one by varying all system configuration parameters over a fixed set of application parameters and one by varying all viable application parameters over a fixed system configuration parameters. From these two profile data sets, the data points for an arbitrary system configuration and an acceptable set of application parameters are computed using linear interpolation.

Another issue is that some hardware platforms only support very coarse power/energy monitoring for the executing application. For example, ODROIDs have power sensors that only refresh about 3-4x per second. We work around this issue in PEMU by profiling an architecture equipped with fine-grained power sensors of higher refresh rate, like the x86, and then rescaling the collected data to map to the data of similar hardware that is being emulated.

After being scaled if necessary, the profile data are stored and indexed in a SQL database. Statistical properties of the data are extracted in dynamic views providing a basis for introducing randomness into testing and development. Running in the emulated environment, system calls made by the adaptive control software to monitor the behavior of the application in question are carried out by PEMU via appropriate queries to the database and interpolation thereafter if required. For any particular application with a specific application configuration, a specific input unit of the input stream and a specific system configuration, this schema provides the amount of time and energy used to process that particular input unit.

PEMU uses SQLite as the SQL database engine. Figure 3 is the Entity Relationship (ER) diagram of the major tables in the PEMU database. At the heart of the database is the table

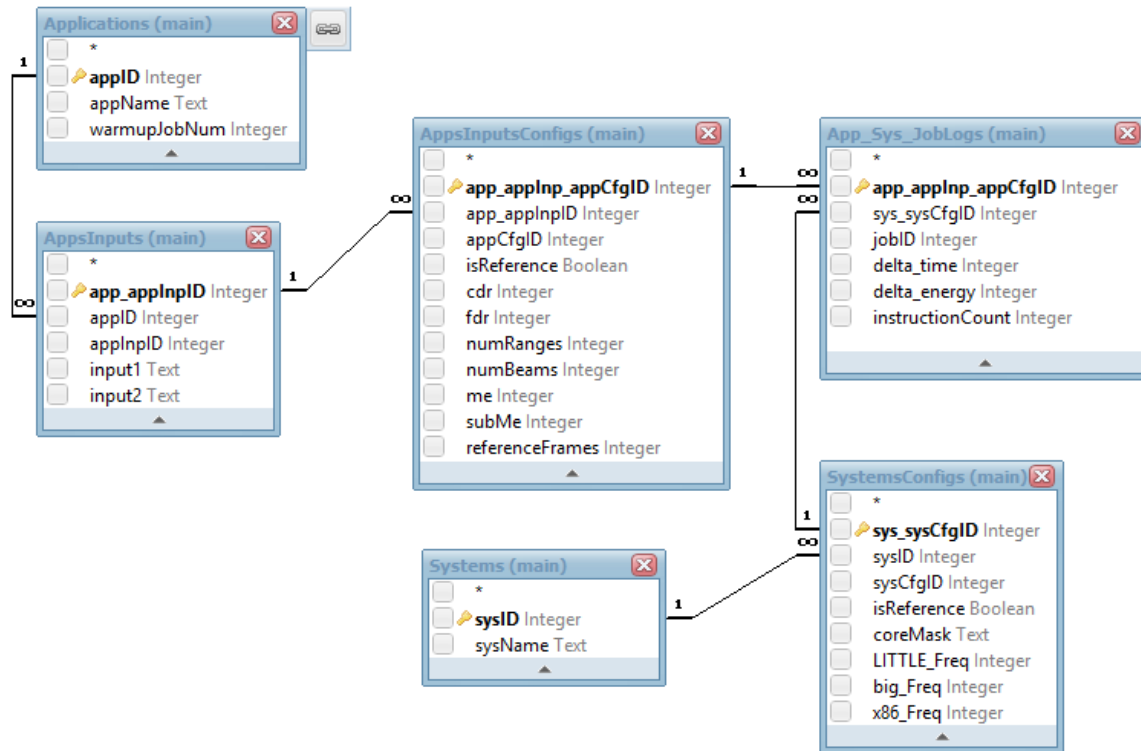


Figure 3: PEMU ER.

App_Sys_JobLobs table. It stores “log” entries that are used to emulate application and system behavior, based on a combination of keys:

- app_appInp_appCfgID, which uniquely identifies the application, its inputs and its parameters,
- sys_sysCfgID, which uniquely identifies the hardware system and its configuration parameters,
- jobID, which uniquely identifies the input unit in the input stream.

Together, these three attributes form the primary key that serves to uniquely identify the amount of time (delta_time) and energy (delta_energy) used to process a given input unit.

PEMU Client

To enable random external perturbations to the system, PEMU uses asynchronous communication via a TCP connection. We provide a static C library called `libpemu_client.a` with the following public interface to interact with an application running under PEMU.

When executed, `pemuSendMessage` will block until a reply message is received back from PEMU and will return the reply message in the second argument of the function call.

Table 1: libpemu_client API

Function	Description
pemuClientGetBufferSize	Gets the buffer size
pemuClientInit	Initialize the TCP socket connection with the Proteus Emulator (PEMU). parameters: 1. serverIP: the IP address of the server 2. port: the port number 3. bufferSize: the maximum buffer size for all communications.
pemuClientFinish	Close TCP socket connection setup by pemuClientInit. Pre-condition: pemuClientInit(...) has been called.
pemuSendMessage	Send a message (mess) to the server, block and get back a result string (result). Return the length of the result string. Pre-condition: pemuClientInit() has been called. parameters: 1. mess: message of maximum length given by pemuClientGetBufferSize() sent to PEMU 2. result: message of maximum length given by pemuClientGetBufferSize() received back from PEMU

PEMU Test Adapter

To facilitate uniform testing and evaluation across all BRASS performers, the evaluator specified a set of RESTful APIs for communication with its Test Harness. Table 2 shows the interface of the PEMU Test Adapter component, which consists of two functions. Rice Proteus need not implement a REST service and only acts as a REST client using the following setup.

- The evaluator creates a /test directory and populates it with a configuration file that describe.
- The evaluator then runs a startup script provided by Rice Proteus which parses the data file, generates appropriate scripts to perform the specified scenario, and runs them.

For each test run, the application running under PEMU establishes connection with the test harness by calling testAdapterInit and invokes testAdapterHttpPost to post appropriate requests in JSON format to the test harness, allowing the evaluator to witness the intent element acting within the system.

At startup, when the Proteus system is ready to run a test, it sends the DAS_READY message to the test harness. In case of an error is encountered at any stage, it sends the DAS_ERROR message to the test harness. The STATUS message is sent to the test harness when the TCP message “get status” (see message table in the above Online API section) is executed. The DONE message is sent after completion of a test run.

Table 2: PEMU Test Adapter Interface

Function	Description
testAdapterInit	Obtain the endpoints to communicate with the test harness server from the appropriate environment variables.
	Send an http POST request to the server with a JSON payload. parameters:
	1. url: the server URI
	2. json_body: the JSON payload
testAdapterHttpPost	3. message: text PEMU can use to print the requests sent to the http server
	return:
	1. U_OK if POST request is successful, otherwise return an appropriate error code

3.1.1.2 PEMU API Layer

The PEMU API layer provides access to the emulator state by supporting the public interface described in Table 3.

An application running under PEMU works at a high level as follows.

1. Call pemuInit, passing it the application's name and application specific data to
 - (a) properly initialize itself with PEMU,
 - (b) establish: the TCP client connection, the PEMU database connection, and the connection with the test harness server.
2. For each input unit in the input stream do
 - (a) Process the input unit;
 - (b) Call pemuReportFinishedInput to report the process of the input unit by
 - i. updating the emulator state,
 - ii. checking and handling message from the client TCP connection, calling pemuAPI to process and report the processing of the task.
3. Call pemuFinish to tear down all communication and database connections.

Figure 4 depicts an example run of the X264 video encoding application under PEMU.

Figure 5 depicts an example of a client application interacting the X264 app and PEMU by sending messages such as processing some number of inputs, getting its current status.

3.1.2 Task 1.2: Emulation Platform and Cost Modeling

The goal of this task was to provide power and performance models that could be integrated into the Proteus emulator, enabling the TA2 and TA3 teams to make progress before the hardware platform was complete. Our initial proposal was two-pronged:

Table 3: PEMU Public Interface

Function	Description
pemuInit	Initialize an application with PEMU, establish the TCP client connection, the PEMU database connection, and the connection with the test harness server. Pre-condition: set the environment variable PEMU_DB to the full path name of the pemu database. parameters: 1. application: the name of the running application 2. applicationData: data specific to the application
pemuFinish	Close the client TCP connection, the database connection and the connection to the test harness.
pemuReportFinishedInput	The streaming application reports the processing of one input.
pemuAPI	Setting or getting an internal element the private emulator parameters: 1. directive: the name of the directive 2. apiArguments: list of arguments specific to the directive 3. resultStyle: the format of the returned string, one of {Enriched, Simple}.

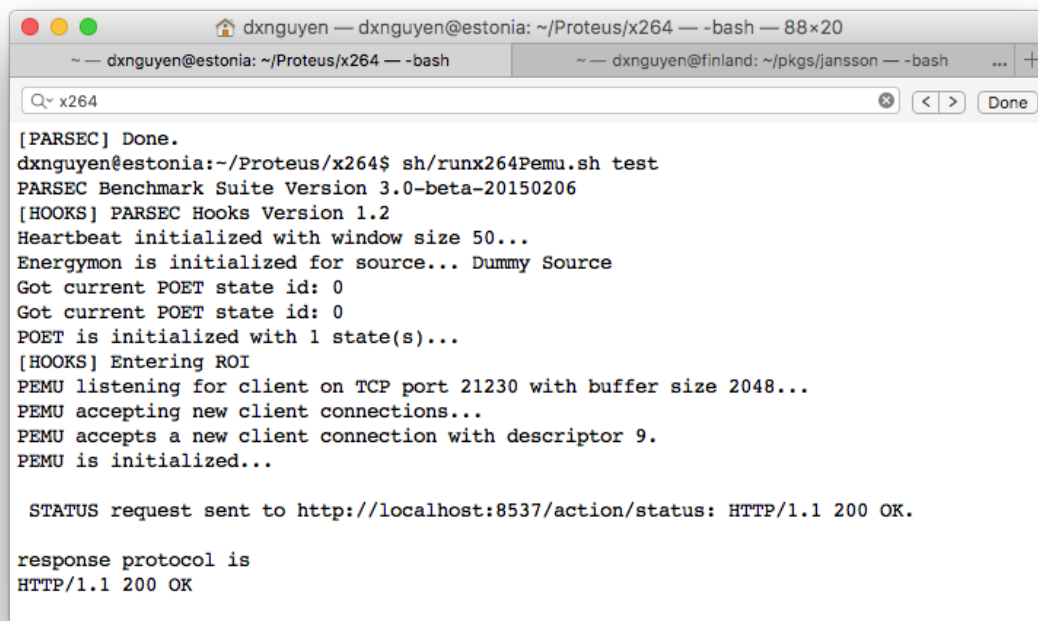
1. to acquire an embedded system that would form the base of our hardware platform and characterize its power and performance, and
2. to implement hardware subcomponents in FPGAs to estimate their characteristics.

After exploring several options and carefully evaluating the Phase 1 requirements for our emulator, we settled on an alternative approach based on capturing and replaying data. The sections below describe our preliminary investigations and the approach we used.

3.1.2.1 Preliminary Investigation

In the beginning of the project, we evaluated several candidate embedded systems that could form the foundation of the Proteus Hardware Platform. Options that were seriously considered include: the Phenox2, the AscTec Atomboard v3, the AscTec Mastermind with CPU option 3b, and the ZedBoard SDR II. Ultimately, we determined that the ideal platform was a Xilinx product that had been announced but was not yet widely available. The system selected was the ZCU102 Evaluation Board based on the newly-released Zynq Ultrascale+ MPSoC (multiprocessor system-on-a-chip). The Zynq Ultrascale+ includes several different types of embedded processing cores, a generous amount of programmable logic, and several hardware accelerators in a tightly integrated single-chip package. This design will provide our system with numerous opportunities for adaptation and exemplifies the types of processors we expect to see on future drones. Unfortunately, the ZCU102 did not become available until late in Phase 1.

Concurrent with the search for an embedded platform, we began evaluating options for our emulator implementation. We created an initial prototype based on a detailed architectural simulator (Graphite from the Carbon group at MIT). This prototype was able to estimate the power and performance characteristics of any application running on an embedded processor model given detailed models of the processor subcomponents (*e.g.*, arithmetic logic units, floating-



```
[PARSEC] Done.
dxnguyen@estonia:~/Proteus/x264$ sh/runx264Pemmu.sh test
PARSEC Benchmark Suite Version 3.0-beta-20150206
[HOOKS] PARSEC Hooks Version 1.2
Heartbeat initialized with window size 50...
Energymon is initialized for source... Dummy Source
Got current POET state id: 0
Got current POET state id: 0
POET is initialized with 1 state(s)...
[HOOKS] Entering ROI
PEMU listening for client on TCP port 21230 with buffer size 2048...
PEMU accepting new client connections...
PEMU accepts a new client connection with descriptor 9.
PEMU is initialized...

STATUS request sent to http://localhost:8537/action/status: HTTP/1.1 200 OK.

response protocol is
HTTP/1.1 200 OK
```

Figure 4: X264 App Running Under PEMU

point units, register files, caches, etc.) which we had planned to create by implementing each component using FPGAs. However, it became apparent that this approach was too detailed and expensive for our needs. The detailed simulation was very slow. To overcome this problem, we adopted a coarser-grained approach that was much faster and sufficiently accurate to support work on TA2 and TA3 tasks and to accommodate evaluation by the Lincoln Lab team.

3.1.2.2 Revised Methodology

Based on the early findings above, we carefully reassessed the requirements for our emulator and isolated two key facts:

1. Phase 1 only required supporting the challenge problem applications, not any arbitrary application; and
2. we only need to provide power and performance results at the same granularity that our runtime system requests them, not at a detailed per-instruction granularity.

This immediately suggested an approach where we capture performance and power data from runs of our challenge applications on actual hardware and replay the appropriate data when emulating each application.

To implement this scheme, we piggyback on the normal queries that the runtime system makes anyway. The runtime system uses a library that queries the platform to calculate the elapsed time and energy used since the previous query. It calls this library every time it completes a work unit (or "job") so that the data returned represents the time and energy needed to complete that particular job. (It uses this information to determine if the application is meeting its

```
Proteus — dxnguyen@estonia: ~ — -bash — 90x49
~/Documents/Proteus — -bash ... ~/Documents/Proteus — dxnguyen@estonia: ~ — -bash +
dxnguyen@estonia:~/Proteus/pemu-client$ ./default-client
Client socket connected using TCP port 21230 with buffer size 2048.
Client Initialized.

>> process 20
Processing 20 input(s).

>> get status
{
  "TIME": "2017-03-09T19:54:14Z",
  "ARGUMENTS": {
    "Application": "x264",
    "Architecture": "ARM bigLITTLE",
    "state": "Online",
    "emulationMode": "Scripted",
    "energy": 148876208,
    "runningTime": 27100905190,
    "numberOfProcessedInputs": 20,
    "optimizationConstraint": 0.90000000000000002,
    "applicationKnobs": {
      "ME": 16,
      "subMe": 7,
      "referenceFrames": 5
    },
    "systemConfigurationKnobs": {
      "utilizedBigCores": 4,
      "utilizedLittleCores": 0,
      "utilizedBigCoreFrequency": 2000000,
      "utilizedLittleCoresFrequency": 200000
    },
    "scenarioKnobs": {
      "availableBigCores": 4,
      "availableLittleCores": 0,
      "maximalBigCoreFrequency": 2000000,
      "maximalLittleCoreFrequency": 1400000
    },
    "applicationAccuracy": 9897457.0,
    "verdictComponents": {
      "windowPerformance": 0.73798273008902415,
      "windowError": 0.18001918878997319,
      "MAPE": -1.0,
      "verdictEnergy": -1.0
    }
  }
}

>> process 100
Processing 100 input(s).
```

Figure 5: A Client App Interacting With x264 and PEMU

performance or power intent). For modeling, we run our system on real hardware and simply record the values returned by each call to the library. When running under emulation, these calls are intercepted and the recorded data is returned. To make the emulation more realistic, random noise is injected into the returned values to simulate normal run-to-run variation.

The problem with this scheme, as described so far, is that it captures data only for the particular combination of knob settings that was being used for each job. Under normal operation, the runtime system periodically adjusts the knobs to maintain the user intent. Therefore, to properly emulate the hardware platform, we must respond to each query with the correct time and energy for the particular combination of knob settings that are currently in effect.

For Phase 1, with only two applications and a modest number of knobs, it is feasible to simply rerun the applications with every possible combination of knob settings. During these runs, the runtime system is disabled to prevent it from making dynamic changes to the settings. Thus, each run collects data for all of the jobs given one particular configuration of knobs. The results from all of these runs are collected into a database for use by the emulator. Whenever the emulator intercepts a call to the performance/energy library, it indexes into the database using the current knob settings and retrieves the appropriate time and energy values.

Because the ZCU102 board was not available until the end of Phase I, we selected an alternate embedded platform to use for the evaluations in this first phase. The selected board (an ODROID-XU3 from HardKernel) uses the same ISA (ARM) and has the same number of large cores (four) as the ZCU102. The key differences are that the individual cores are not as powerful as the ZCU102 and the ODROID lacks programmable logic. Neither of these differences were important in Phase 1: the weaker cores simply make this board more representative of current, rather than future, embedded processors and the use of the programmable logic was not planned until Phase 2 and Phase 3 anyway. Starting in Phase II, the emulator was updated to support the ZCU102 board.

3.1.2.3 Enhancements

One limitation of the initial scheme was that it could not properly model the effects of different input data in some applications. The first challenge application for Phase I/II (radar) performs the same calculations on each input pulse independent of its value. Therefore, the database we construct will still be valid even if the input is changed.

However, the computation for our second Phase I/II/III challenge application (x264) is highly data-dependent. Changing the input video stream can drastically change the amount of computation that must be done for a given frame because the video encoding algorithm examines successive frames to find redundant information both within and between frames. Hence, our record-and-playback approach to emulation only works for a fixed input stream. If a different input stream is used, the computational load of each individual job (*i.e.*, frame) will be different. To address this problem, we changed the way we indexed into our database. Instead of simply looking up the data for a particular job number, we collected data for each input processed by the application, and made the input part of the key used to retrieve emulation data.

3.1.2.4 Conclusion

Based on early development of our hardware platform and emulator, we determined that the modeling approach described in our proposal was too computationally expensive and too elab-

orate for the project's needs. Instead, we designed a new approach based on alternate hardware and course-grained capture of data. This solution yielded high-performance emulation with results that reflect what we observed on the final hardware platform.

3.1.3 Task 1.3: Machine-Learning Drone Design and Validation

The prototype machine learning drone platform is an important component of the Proteus project. This platform allows us to run experiments on real hardware that is representative of the computing infrastructure that might be found in an autonomous flying vehicle. In Phase I, we established the system requirements for this platform and evaluated several options for the hardware. We ultimately selected the Xilinx ZCU102 FPGA evaluation board and received the hardware late in Phase I. In Phase II, we focused on the software components of the platform, including board firmware, operating system, libraries and compilers. We began Phase III with some updates to the compilers and support for machine learning but the platform was stable and in-use for most of the phase.

3.1.3.1 Low-level Software (Firmware and Operating System)

The foundation of the platform software system is provided by the Xilinx PetaLinux tool. This tool can generate a bootable image for the ZCU102 that contains ATF (ARM Trusted Firmware), a BootROM, FSBL (First-Stage Boot Loader) and a light-weight Linux installation. This image is placed on a standard SD card that is then inserted into a socket on the ZCU102 board. When the board is turned on, it reads the information from the card to setup its firmware and boot the Linux operating system.

Unfortunately, the Linux distribution provided by PetaLinux was not sufficient for our needs. Therefore, our first step was to modify the SD card image to replace the provided Linux partition with an image containing a full Ubuntu 16.04 distribution. This version of Linux was chosen because it was mature and stable and had been used for the rest of Proteus software development. Although it did not require any technical innovation, the replacement of the Linux partition was non-trivial as it was necessary for us to manually modify several configuration files in the image and the process was not documented or supported.

3.1.3.2 Swift Compiler

Once the ZCU102 board was up and running with an appropriate operating system, we turned our attention to the specific software infrastructure needed to support Proteus. While our Phase I challenge problems and runtime system had been based on the C/C++ language, a major part of our Phase II effort was the development and use of FAST which is based on the Swift programming language. Therefore, the most critical piece of infrastructure needed was a compiler for Swift.

One of the reasons that we chose Swift as the basis for FAST was the availability of an open-source compiler. Apple originally developed Swift internally for use on their iOS systems but has, more recently, released and supported an open-source version for Linux. Since most iOS devices use processors based on the ARM architecture, we did not anticipate major problems

running Swift on our ARM-based Xilinx ZCU102 boards. Unfortunately, we were overly optimistic and soon found that ARM was not well supported for Linux systems. As a result, we spent a lot of effort during Phase II setting up and maintaining this piece of infrastructure.

In Phase II, after many attempts to get Swift version 3.1 working on ZCU102 board, we eventually realized that there were critical pieces of the compiler that were simply not yet implemented for the 64-bit ARM architecture (called “aarch64”) that our prototype uses. Our solution was to fall back on the 32-bit compatibility of our ZCU102 and use a pre-built version of the Swift compiler intended for Raspberry Pi systems. Although this allowed us to make progress, it was difficult to port and maintain 32-bit versions of Proteus code and the 3rd-party libraries we depended on.

Early in Phase III, Swift 4.0 was released and we decided to try for a 64-bit version once again. Between version 3.1 and 4.0, significant progress had been made by other developers on aarch64 support. By collaborating with an online community of people who were using Swift on ARM platforms, we were able to fix some remaining bugs and eventually get a 64-bit version of Swift 4.1 working on our platform. Although this required substantial effort on our part, we were able to greatly reduce the effort we spent on porting and maintenance and contribute our fixes back to the open-source community.

3.1.3.3 Other Libraries and Dependencies

The biggest disadvantage of using a 32-bit version of the Swift compiler was that it required all libraries and dependencies to be 32-bit as well. In Phase II, we expended considerable effort getting 32-bit versions of all the 3rd party Swift packages we needed working on our system. Fortunately, after the change to 64-bit the use of 3rd party software was much easier. For example, we were able to easily add the Tensorflow libraries to our system to support of the on-line machine learning component of Proteus.

However, we still needed to update some packages to use the specific hardware interfaces available on our platform. For example, the method of accessing the energy monitoring circuits on our new platform was slightly different than any previous platform we had used. In general, the specific number and type of sensors on each platform is different and therefore requires some specialization. We created a new energymon “implementation” for the ZCU102 board and contributed this back to the open-source community. We also needed to make small changes to our code that interfaces with system knobs. For example, changing the core frequency requires writing to different system interface files. We these final changes, the platform was complete and was used heavily by the Proteus team for the remaining 12 months of the project.

3.1.4 Task 1.4: Interfacing Machine-Learning Drone Design with Emulator

The interface of the Proteus hardware platform and the emulator is described in Section [3.1.2](#).

3.1.5 Task 1.5: Machine-Learning Drone Laboratory Validation Using TA3 and TA2 Outcomes

In Tasks 1.3 and 1.4, we implemented a basic prototype hardware platform and began using it for our challenge problems. This task built on that work, improving the platform and fully implementing all challenge problems on it.

3.1.5.1 Platform Improvements

Although the work done in Task 1.3 (Section 3.1.3) was sufficient to allow initial development to proceed on the ZCU102 prototype platform, it became apparent late in that task that this was not sufficient for the long-term. Specifically, we encountered bugs in the version of the Swift compiler we were using (3.1.1) and found that maintaining support for 32-bit and 64-bit compilers was an inefficient use of resources. Since the bugs we encountered had been fixed in later versions of the Swift compiler, we decided to try updating the platform.

Since we had first tried to setup Swift on the prototype platform, a new major version of Swift had been released (version 4) with some substantial changes from the previous version. Although it still did not include official support for our AArch64 Linux platform, other developers had become interested in 64-bit ARM platforms and filled in many of the missing pieces. We identified three key problems that needed to be addressed: missing support for *variadics* (function calls with variable numbers of parameters), several outdated dynamic assertions in the Swift compiler itself, and a lack of AArch64 as a target platform in the Swift Package Manager.

The missing support for variadics was preventing a successful build of the Swift Foundation library, a set of core utilities and interfaces needed by most non-trivial programs. We began researching the 64-bit ARM calling convention (ABI) and implementing this ourselves, but before we could finish someone else submitted a pull request (PR #15174) to the Swift GitHub repository [4]. Rather than duplicate their effort, we applied their patch and (after helping them fix a minor bug) were able to successfully build the Swift Foundation library. The outdated dynamic assertions were simply disabled in the custom build of the compiler used on the Xilinx ZCU102 platform. The runtime errors experienced in tests on this platform did not affect the x86 Linux builds used in the Lincoln Laboratory evaluation infrastructure, or the environment set up internally to mirror that infrastructure.

With a working Swift compiler and core libraries we could theoretically build Swift applications for our AArch64 platform. However, our previous development on FAST and the challenge problems relied heavily on the Swift Package Manager. The Swift Package Manager makes it easy to specify dependencies among our various software components and external third-party packages. It also automatically downloads and builds all the dependencies, plus any packages *they* depend on, *etc.* This greatly simplifies the task of using third-party code and, therefore, it was important for us to fix the missing support for AArch64 in Swift Package Manager. Although the Swift Package Manager code is not well documented, we were able to make the necessary changes by copying the code for other platforms and modifying it slightly for AArch64. After doing this, we contributed our code back to the community of Swift ARM

users we had identified in Task 1.3 [58] so others could benefit from our work.

With these fixes made, we had a fully functional build system for Swift 4.1 on the Xilinx ZCU102 platform. This solved our problem with the bug in Swift 3.1.1 and allowed us to move forward in our Proteus software development. As an added benefit, switching from 32-bit to 64-bit gave us a small performance boost and made it much easier to build new applications and maintain the platform software.

3.1.5.2 Porting FAST and the Challenge Problems

The Phase II and III challenge problems were developed as FAST applications, which means that the executable is a Swift application. To validate the feasibility of developing embedded applications from scratch using this platform, the CP1 for Phase II (SAR, described in Appendix 5.2.3) was implemented as a Swift application, with support from C libraries for signal processing. This design made it possible to use Swift’s Grand Central Dispatch framework to parallelize the application, and resulted in a more compact implementation compared to the SAR implementation evaluated in Phase I. The two other challenge problems (x264, described in Appendix 5.2.3, and CaPSuLe, described in Appendix 5.2.3) were implemented as wrappers for native C and C++ applications, respectively, using Swift’s foreign function interface. This approach proved very successful, requiring modest changes to the legacy code bases, while allowing the Proteus runtime to monitor and control the applications, and orchestrate evaluation using the interfaces to the Lincoln Laboratory test harness.

The incomplete status of the Swift standard library on Linux [6] required custom implementation of certain data structures in the challenge problem applications as part of porting to the Linux/Xilinx platform. The move to Swift 4.1 changed the way in which dependencies are resolved by Swift Package Manager. This change simplified the process for upgrading the version of FAST used by the challenge problem applications (a single, but also demanded version control of the dependency list (“pins”) that Swift Package Manager generates for the challenge problems during a build, to ensure that incompatible versions do not replace the tested versions while during a build.

3.1.5.3 Platform-Specific Interfaces

The final changes required to run our challenge problems on the new prototype hardware platform involved modifying our system interfaces. For example, the method of accessing the energy monitoring circuits on our new platform was slightly different than any previous platform we had used. In general, the specific number and type of sensors on each platform is different and therefore requires some specialization. We created a new Energymon “implementation” for the ZCU102 board and contributed this back to the open-source community [34]. We also needed to make small changes to our code that interfaces with system knobs. For example, changing the core frequency requires writing to different system interface files.

3.2 Technical Area: Intent-Aware Discovery Through Machine Learning

3.2.1 Tasks 2.1 & 3.2: Interface between TA2 and TA3 Innovations

This task focused on defining the abstractions and data structures that allow our offline machine learning system, which discovers quality and cost tradeoffs, to communicate with the online control system, which converts application specified goals into specific application configurations. Without loss of generality, our work assumes that cost is measured in time, and that the control system relies on a machine learned (ML) database capturing the tradeoffs between quality (typically accuracy) and performance of the application to be controlled.

We elide the details of the control system that are not relevant to the interface. At some point, however, the control system must convert a generic *speedup* requirement at time t into one or more application configurations. We call the assignment of time to configurations a schedule. There are typically many schedules that meet a particular performance requirement and we want to find a maximal accuracy schedule. Given a time interval T , a speedup requirement $speedup(t)$ to average over the time interval, and a configuration set of size C , we formalize this problem as:

$$\begin{aligned} &\underset{\tau \in \mathbb{R}^C}{\text{maximize}} && \sum_{c=0}^{C-1} \tau_c \cdot a_c \end{aligned} \tag{1}$$

$$\begin{aligned} &s.t. && \sum_{c=0}^{C-1} \tau_c \cdot s_c = speedup(t)T \end{aligned} \tag{2}$$

$$\sum_{c=0}^{C-1} \tau_c = T \tag{3}$$

$$0 \leq \tau_c \leq T, \quad \forall c \in \{0, \dots, C-1\} \tag{4}$$

where a_c and s_c are configuration c 's estimated accuracy and speedup; τ_c is the time to spend in configuration c . Eqn. 1 simply states that the objective is to maximize accuracy (cumulative over the time interval T). Eqn. 2 states that the average speedup must be maintained, while Eqn. 3 requires the time to be fully utilized. Eqn. 4 simply avoids negative time.

We assume that the values a_c and s_c are not known, but instead estimated by the ML component. In that case, we need to communicate these estimates to the control system to solve Eqns. 1–4. Since this solution is found locally on an embedded device, we must do so efficiently.

To achieve this efficiency we (1) exploit the structure of this particular optimization and (2) encode the learned model in a *performance hash table (PHT)*. Combining these two ideas we solve Eqns. 1–4 in constant ($O(1)$) time.

Kim et al. have analyzed solutions to the problem of minimizing energy while meeting a performance constraint [31] – a problem which is mathematically identical to the problem explored in this task. They observed that there must be an optimal solution with the following properties:

- At most two of τ_c are non-zero, meaning that at most two configurations will be used in any time interval.
- If you chart the configurations in the accuracy loss and performance tradeoff space the two configurations with non-zero τ_c lie on the lower convex hull of the power performance tradeoff space.

We use these two facts to construct a constant time algorithm for finding the optimal solution online.

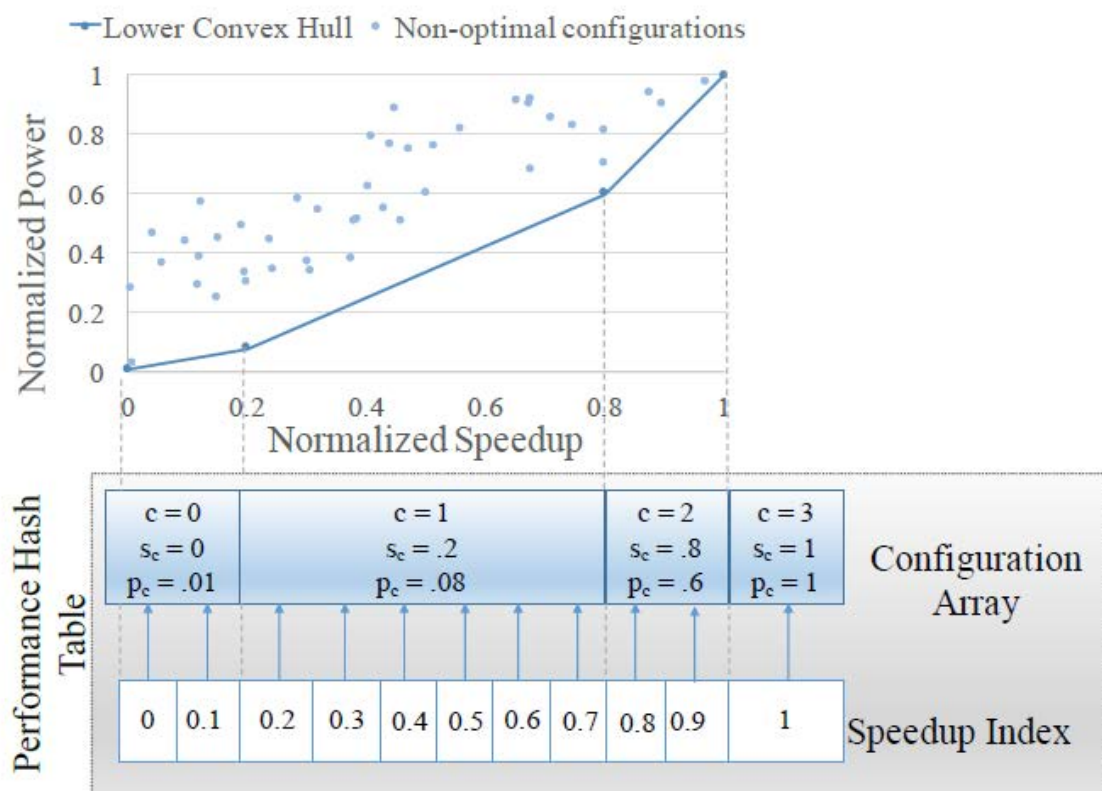


Figure 6: Data structure to efficiently convert required speedup into a configuration that minimizes energy – the same data structure can be used to convert speedup into a maximum accuracy (or minimum accuracy loss).

The PHT (illustrated for performance and power tradeoffs in Figure 6) only contains points on the lower convex hull of the power/performance tradeoff space. It consists of two arrays: an array of pointers that point into a second array, which stores the configurations on the convex hull sorted by speedup. Speedups are computed relative to the base speed which produces the

lowest accuracy; this is done so that speedup is bounded between 0 and 1, bounding the indices into the table. The first table, of pointers has a *resolution* indicating how many decimal points of precision it captures. The example in Figure 6 has a resolution of 0.1. Each pointer in the first table points to the configuration in the second array that has the largest speedup less than or equal to the index.

To find an optimal configuration that meets the required performance the optimizer receives a speedup $speedup(t)$ from the controller. It converts this into two configurations referred to as *hi* and *lo*. To find the *hi* configuration, it clamps the desired speedup to the largest index lower than $speedup(t)$ and then walks forward until it finds the first configuration with a speedup higher than $speedup(t)$. This configuration is *hi*. To find the *lo* configuration, the translator clamps the desired speedup to the smallest index higher than $speedup(t)$ and then walks backwards until it finds the configuration with the largest speedup less than $speedup(t)$.

For example, consider the PHT in Figure 6 and an optimizer meeting $speedup(t) = .65$. To find *hi*, the optimizer indexes at .6 and walks up to find $c = 2$ with $s_c = .8$, setting $hi = 2$. To find *lo*, the optimizer indexes the table at .7 and walks backward to find $c = 1$ with $s_c = .2$, setting $lo = 1$.

Finally, the optimizer sets τ_{hi} and τ_{lo} by solving the following set of equations:

$$T = \tau_{hi} + \tau_{lo} \quad (5)$$

$$speedup(t) = \frac{s_{hi} \cdot \tau_{hi} + s_{lo} \cdot \tau_{lo}}{T} \quad (6)$$

In these equations, $speedup(t)$ is the speedup requested by the controller and s_c are speedups estimated by the learner. By solving Eqns. 5 and 6, we have turned the controller's speedup into a schedule of resource allocations using the models provided by the ML component.

Provided that the resolution is large enough to get a good spread of configurations to indices, the translator will always index the configuration array at most one entry from where it needs to be. Thus, the entire translation process runs in constant time—assuming that the learner is responsible for building the PHT once before passing it on to the control system. This efficiency comes at a cost of memory usage; many of the entries in the speedup index table will point to redundant locations in the configuration array. We think that this is a reasonable trade-off to make in practice as the code that runs on the embedded device must be fast or we risk wasting energy and speed. In practice, we recommend a table of size 100 which provides a sufficient resolution and is not too wasteful of space.

3.2.2 Task 2.2: ϵ Approximate Control Theoretic Framework and Assurance Monitor Design

3.2.2.1 ϵ Approximate Control Theoretic Framework

In order to build a framework for writing embedded streaming applications that dynamically adapt to changes in the operating environment or in user intent by trading inexactness for better performance or energy utilization, we proposed to develop the theoretical foundations underlying the corresponding control problem. Generally, inexact computing aims to compute good

solutions that require significantly less resources such as energy than corresponding exact solutions. While inexactness was originally motivated by concerns derived from technology scaling and Moore’s law, there is no formal or foundational framework for reasoning about this novel approach to designing algorithms.

Cost–quality tradeoff, Symmetries

As a first foundational step, we have formulated a relationship between the quality of computing the value of a Boolean function and the energy needed to compute it in a mathematically rigorous and general setting. On this basis, one can study the tradeoff between the quality of the solution to a problem and the amount of energy that is consumed.

To this end, we introduced a computational model to classify problems based on notions of symmetry inspired by physics. We found that some problems are symmetric in that every input bit is, in a sense, equally important, while other problems display a great deal of asymmetry in the importance of input bits. We believe that our model is novel and provides a foundation for inexact computing. In this context, we can show that asymmetric problems allow us to invest resources favoring the important bits – a feature that can be leveraged to design efficient inexact algorithms. In contrast, we can also prove that the best inexact algorithms for symmetric problems are no better than simply reducing the resource investment uniformly across all bits. Akin to classical theories concerned with space and time complexity, we believe the ability to classify problems as shown in our work will serve as a basis for formally reasoning about the effectiveness of inexactness in the context of a range of computational problems with energy being the primary resource.

Inexactness, Model Formulation

Assuming that we are given an asymmetric problem, the next theoretical question that must be addressed is how to achieve the most accurate result for a given energy budget. In other words, given an energy budget, how can we optimally distribute this energy among the different bits. On some functions the optimal distribution that we call a *naive solution* is simply a uniform distribution. On other functions, computing the optimal distribution, requires non-trivial *smart* assignment of computational resources. We identify the ratio between smart and naive solutions as a parameter, that we call the *influence ratio*, for function classifications: the smaller the influence ratio gets, the better the smart solution is.

Formally we start with a given function $f : \{0, 1\}^n \rightarrow \mathbb{N}$. We assume a *reader* which is a function $\mathcal{R} : \{0, 1\}^n \rightarrow \{0, 1\}^n$ that is a function that “scrambles” the data such that the output of f is different than the output of $f \circ \mathcal{R}$. The reader we assume in this work is *adversarial* in a sense that we cannot make any assumption about the obfuscation that the reader provokes. In addition, we also briefly touch a *stochastic* reader on which actions we are more informed.

To handle the obfuscation caused by the reader, we assume that we are given an energy amount $\mathcal{E} \geq 0$ that can be divided to n elements $\vec{e} = (e_1, \dots, e_n)$ (then $\sum_{i=1}^n e_i = \mathcal{E}$), and a transformation function $\mathcal{F} : \vec{e} \rightarrow [0, 1]$, such that if we denote $p_i = \mathcal{F}(e_i)$, then we make the reader obfuscate the input bit with probability p_i . An example of such transformation can be $\mathcal{F}(e_i) = 1/2^{e_i}$: the more energy that is allocated to that bit, the less chances the bit will flip to the incorrect value.

We define $\vec{p} = (p_1, \dots, p_n)$. We note that $\vec{p}$ is not a probabilistic distribution, but each $(p_i, 1 - p_i)$ is a probabilistic distribution on its own. This setup is shown on Figure 7.

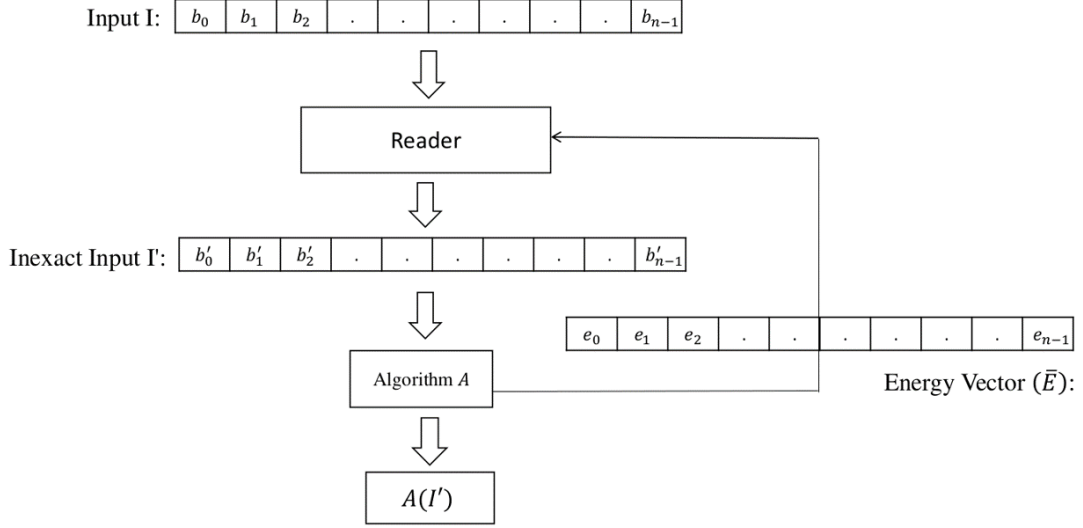


Figure 7: Reader Scrambling Input

Our formal model is motivated by the idea that the more energy we put in a bit i , the less chance that the reader gives an erroneous output on the bit i . By observing f and distributing the cost $\mathcal{E}$ we can decide which bits are less vulnerable to change the outcome of f , so they can be flipped with higher probability and give an inexact answer in return for saving the energy that would be required to “control” these bits. This model in which we can control this distribution of $\mathcal{E}$ in exchange for inexact results is called *inexact computing*.

In the most general setting that we will be investigating, this notion of inexactness will be the basis for solving *constrained satisfaction problems* (CSPs) [43] and approximate solutions. A specific set of constraints that naturally model our setting is shown below. We have constraints on energy and execution time, controlled by parameters or “knobs” of Proteus (K_1 and K_2), whereas a constraint on the minimum acceptable quality of the solution (f_Q).

$$\begin{aligned} f_E(K_1, K_2) &\leq E_T && \text{(Energy Constraints)} \\ f_{RT}(K_1, K_2) &\leq RT_T && \text{(Runtime Constraints)} \\ f_Q(K_1, K_2) &\geq Q_T && \text{(Quality Constraints)} \end{aligned}$$

Since we are looking at a dynamically evolving environment in the context of Proteus, our model is streaming a sequence of such constraint satisfaction problems, and it is well known that, subject to widely accepted theoretical assumptions, the best known solution for such problems in the context of approximation are *semidefinite programming techniques* (SDPs) [43]. This concept is illustrated in Figure 8a, whereas our intention to inject our formal models for inexactness, described above, into the SDP framework for approximately solving CSPs is shown in Figure 8b.

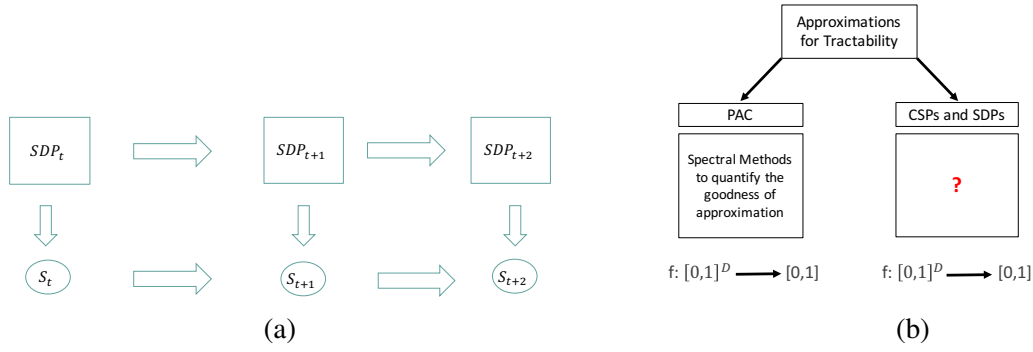


Figure 8: (a) A Streaming Model For Semidefinite Programs (b) Injecting Inexactness Into SDP-Based Approximation Algorithms for CSPs

3.2.2.2 Assurance Monitor Design

Assurance monitoring was investigated in several contexts during the project.

The general framework for assurance monitoring that was developed in Phase II is described in Section 3.2.5. Its software is available in the MIT GitHub [29].

Further work on the predictive modeling aspects of assurance monitoring were developed in Phase III, and focused on chaotic dynamical systems [15] as a challenging dynamical systems prediction benchmark. This work was included as an element of the Phase III demonstration (Section 4.2.2.3).

3.2.3 Task 2.3 & 2.5: Resilient and Adaptive Memory Models

The goal of the BRASS program was to design systems with long-term resilience and survivability. An important aspect of this is considering how software will operate on future generations of hardware. Porting a parallel program to a new upgraded embedded hardware platform is notoriously difficult. This portion of the Proteus project examined reasons for this difficulty and proposed a new approach to designing hardware and software to address them. Because this work addresses a different aspect of resilience than the rest of the Proteus project, most of it was done independently, assuming different hardware and software environments. However, we have tied it back into Proteus by evaluating performance on two of the Proteus challenge problem apps.

3.2.3.1 Why Programs Break Down on New Embedded Platforms

Embedded programs are written in a mixture of assembly language and low-level C for performance reasons. This style of software is not resilient to changes in hardware platform for the following reasons:

1. *Ad Hoc Mechanisms for Preventing Instruction Reordering in Assembly Codes:* Although the assembly code constitutes only a small fraction of the overall program, it contains ISA-specific fence instructions that prevent the processors from reordering instructions at the

points of inter-thread synchronization. These fences are essential to maintain the correctness of the program. Though extra fences do not change functionality, they can hurt performance. The semantics of the fences are defined by the *memory model* and different ISAs support different memory models. Even within the same ISA, sometimes new fences are introduced for performance reasons, e.g., ARMv8 introduced store-release and load-acquire instructions which have implicit fences [8]. Therefore, it is difficult to write correct and efficient assembly codes and keep it up-to-date with the changing platform.

2. *Compiler Optimizations*: An upgraded compiler can transform programs differently, which can inadvertently result in different orderings of instructions. For example, a programmer expects instructions with data dependencies to be executed in order, and may have some code fragments just to enforce ordering. However, if the compiler determines a code fragment to be redundant, it may eliminate it and inadvertently permit some instructions to be reordered.
3. *Lack of Precise Definitions for Weak Memory Models*: A memory model should specify unambiguously all the legal behaviors of a given program. Such a definition exists for sequential consistency (SC) [32] but unfortunately SC is not used by any ISA. Intel/Sparc-TSO is a widely used memory model with a precise definition. However, many companies, notably ARM and IBM, have pursued weaker memory models because they allow more flexibility in processor implementations. Unfortunately, the ISA manuals of ARM and POWER are unclear in the sense that they only specify permitted instruction re-orderings but not the value a load may return. The lack of clear definitions has forced the researchers to formalize these weak memory models to conform to empirically observed behaviors. Such formalizations are brittle, because it is impossible to enumerate all the behaviors of a processor empirically. In fact, the published formalization of POWER memory model [48], which is co-authored by an IBM employee, turns out to forbid a behavior which was later observed in POWER7 processors [3]. Furthermore, the processor vendors are not obliged to obey these formal models in the future implementations; they only need to follow the less-than-precise definitions in the ISA manuals. Thus, it is doubtful that such memory models would be resilient to the future changes in commercial processors.

Another drawback of lack of clear definitions is that hardware may evolve in the “wrong” direction and generate unexpected behaviors, which may affect the correctness of software. For example, in many concurrent algorithms a thread may first read a shared variable, then do some work, and finally read the variable again to check whether its value has changed. The basic assumption behind this approach is that the two loads of the same variable are executed in order. However, the ARM Cortex-A9 processor allows two loads for the same address from the same thread to be executed out of order [7]. To prevent this unexpected behavior, an extra fence needs to be inserted to get the desired behavior.

Addressing these problems in a robust way will require changes to future software and hardware. By using appropriate programming practices, the application programmer will be insulated from these concerns and their source code will continue work properly when recompiled for future systems. However, if we are not careful, this will simply shift the burden to compiler and library developers and result in tools that are just as brittle. If we also change the memory model used by the hardware, we can create a much more robust system that does not require changing the synchronization instructions from one generation to the next.

3.2.3.2 Programming in the C++11 Language Model

In order to make the software program resilient, the correctness argument should not depend on whether the processor could reorder two instructions or whether certain compiler optimization have been disabled for some variables. The program should be written under a unified programming language model, and the responsibility of inserting fences should lie with the compiler. The compiler should also refrain from optimizations that destroy the semantics of the concurrent language being used. We think that the C++11 language and associated model [51] are a good candidate for programming embedded systems because it provides a set of axioms that can be checked to determine the legality of a program behavior. C++11 provides its own architecture-independent primitives to enforce ordering.

Here we briefly explain the C++11 primitives, and then show a programming example using C++11. In C++11, a variable could be either a normal non-atomic variable or an atomic variable. A variable should be declared as `atomic` when there is data race to access it, i.e., the variable is used for synchronization. Non-atomic variables can be accessed by normal loads and stores. There must not be any data race on a non-atomic variable; otherwise the program behavior becomes undefined. Accesses on atomic variables can be associated with different ordering tags. Accesses that do not need to be ordered with others can be tagged as `relaxed`. Loads can be tagged as `acquire` and stores can be tagged as `release`. The communication from a store-release to a load-acquire forms a synchronization. C++11 also allows an access to be tagged as `sc`, which implies the strongest ordering, and all `sc` accesses can be put in a total order.

As an illustrative example, consider the C++11 code for inter-thread message passing shown in Figure 9a. The program has two threads, a producer and a consumer, which share the data array and signal each other using the shared `flag` variable. The producer thread first writes the data array and then sets `flag` to inform the consumer thread that the data is ready. The consumer thread first spins on `flag` until `flag` is set, and then reads the contents of the data array. Since `flag` is used for synchronization, we declare it as `atomic` (line 1), and we use C++11 primitives, store-release and load-acquire, to set and check the `flag` respectively (lines 6 and 10). The data array is declared as normal non-atomic variable because the proper use of synchronization using the `flag` rules out any races in accessing data. In the C++11 model, it is guaranteed that the consumer thread will get the data written by the producer thread. It should be noted that the semantics of the store-release and load-acquire accesses are independent of the platform, and that it is the compiler's responsibility to emit ISA-specific fences and stop certain optimizations on the code.

As a comparison, Figure 9b shows the code written in the old C standard for Intel-x86 and POWER processors. Variable `flag` is declared as `volatile` to stop the compiler from optimizing accesses to it. There is, however, no clear definition of what optimizations are disabled by the `volatile` declaration; the effect of `volatile`, in all likelihood, is going to be implementation-dependent. Additionally the programmer manually inserts assembly fences (lines 5, 7, 14 and 16). For Intel-x86 machines, no hardware fence is needed, but the programmer may need to insert software fences (lines 5 and 14) to stop the compiler from reordering the accesses on data and `flag` in both threads. For POWER machines, programmers need to insert a `lwsync` fence (line 6) in the producer thread to keep the ordering of stores, and to insert an `isync` fence (line 16) to prevent the reordering of loads in the consumer thread. Programmers also need to instruct the compiler not to do certain optimizations. All this requires the

```

1  int data[100]; atomic<int> flag = 0;
2  void producer_thread(int x) {
3      for(int i = 0; i < 100; i++)
4          data[i] = foo(i, x); // write
                                data
5      // store release to set flag
6      flag.store(1, memory_order_release)
7      ;
8  }
9  int consumer_thread(int y) {
10     // load acquire to wait for flag
11     while(
12         !flag.load(memory_order_acquire)
13     );
14     int s = 0;
15     for(int i = 0; i < 100; i++)
16         s += bar(data[i], y); // read
                                data
17     return s;
18 }

```

(a) C++11

```

1  int data[100]; volatile int flag = 0;
2  void producer_thread(int x) {
3      /* ... write data ... */
4      #ifdef __x86_64__ // fence for Intel-
                          x86
5          asm volatile("" : : : "memory");
6      #elif __powerpc__ // fence for IBM
                          POWER
7          asm volatile("lwsync" : : : "memory"
8                          );
9      #endif
10     flag = 1; // set flag
11 }
12 int consumer_thread(int y) {
13     while(!flag); // wait for flag
14     #ifdef __x86_64__ // fence for Intel-
                          x86
15         asm volatile("" : : : "memory");
16     #elif __powerpc__ // fence for IBM
                          POWER
17         asm volatile("isync" : : : "memory")
18         ;
19     #endif
20     /* ... read data ... */
21 }

```

(b) Old C Standard for Intel-x86 and POWER

Figure 9: Message Passing in C++11 and Different ISAs

programmers to be very familiar with the memory model of each ISA. In contrast, programming in C++11 insulates the programmer from platform-specific issues and makes well written programs in C++11 much more resilient.

3.2.3.3 WMM: A New Resilient Memory Model for Processors

Using C++11 does not sidestep the problem of ISA memory models; it just pushes the burden of interacting with hardware to the compiler. In order for a compiler to map C++11 primitives to assembly instructions correctly, a precise ISA definition is still indispensable. Besides, the ISA memory model should be resilient in the sense that it should allow as much flexibility for future hardware optimizations as is feasible, so that we can keep using the same ISA and the same mapping for C++11 primitives as the platform evolves.

We have designed such a resilient memory model called WMM [68, 69]. WMM has equivalent axiomatic and operational definitions, and the definition of WMM is dramatically simpler than other commercial weak memory models like ARM and POWER. WMM allows all kinds of reorderings except a store overtaking a load (i.e., load-store reordering). Forbidding load-store reordering is in line with the fact that most processors commit instructions in order to implement precise exceptions and interrupts. Dropping load-store reordering is also the key factor in achieving the simple definition of WMM. The variety of reorderings allowed by WMM gives abundant flexibility in hardware implementations. Furthermore, WMM does not enforce dependency ordering, and thus allows hardware to perform a large variety of speculations, such as branch prediction, memory dependency prediction, and even load-value prediction. This feature enables WMM to encompass future microarchitectures with deeper speculations.

Operational Definition of WMM

WMM is defined operationally using *Instantaneous Instruction Execution* (I²E), which is the style used in the operational definitions of SC [32] and TSO [50]. The abstract machine of WMM consists of n atomic processors and an n -ported monolithic memory m (Please see Figure 10).

The monolithic memory executes loads and stores instantaneously. Each atomic processor contains a register state, which represents all architectural registers, including both the general purpose registers and special purpose registers, such as PC. The atomic processor executes instructions instantaneously and in order, so it always has the up-to-date register state. The *store buffer* and the *invalidation buffer* between each processor and the monolithic memory allow instantaneous execution of memory instructions and fences. For example, the store instruction completes its execution by writing the data in the store buffer. Thus, the store buffer holds the local stores that have not been globally advertised, while the invalidation buffer keeps the stale values that could still be visible to the processor. The data moves between these buffers and the memory asynchronously in the background as will be explained shortly. The background movement of data also captures instruction re-orderings of interest. The legal behaviors of a program under WMM are defined as those that can result by running the program on this abstract machine.

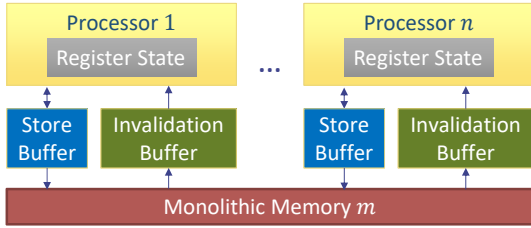


Figure 10: WMM Abstract Machine

C++11 primitives	WMM/RISC-V instructions
Normal Load	Load
Load Relaxed	Load
Load Consume	Load; Reconcile
Load Acquire	Load; Reconcile
Load SC	Commit; Reconcile; Load; Reconcile
Normal Store	Store
Store Relaxed	Store
Store Release	Commit; Store
Store SC	Commit; Store

Figure 11: Mapping C++11 to WMM

We have formally defined the operations of the WMM abstract machine in Figure 12. Each operation consists of a *predicate* and an *action*. The operation can be performed by taking the action only when the predicate is true. Only one operation (either instruction execution or a store buffer dequeue) is performed at a time in the whole system. The choice of which operation to perform is non-deterministic.

If the instruction is not a memory instruction, it simply modifies the local register state of the processor. A store is executed by inserting its $\langle \text{address}, \text{value} \rangle$ pair into the local store buffer instead of writing the data in memory. A load first looks for the load address in the local store buffer and returns the value of the *youngest* store for that address. If the load address is not found in the store buffer, it either reads the value in the memory or any stale value for the load address in the local invalidation buffer. The choice between reading the monolithic memory and the invalidation buffer is non-deterministic. Instead of executing an instruction, we can perform a *background* operation, which removes the *oldest* store for *some* address from a store buffer, writes it into the monolithic memory, and enters the original memory value, i.e. the stale value, into the invalidation buffer of every other processor. It should be noted that whenever a

- **WMM-Nm** (non-memory execution)
Predicate: The next instruction of a processor is a non-memory instruction.
Action: The instruction is executed by local computation.
- **WMM-Ld** (load execution)
Predicate: The next instruction of a processor is a load.
Action: Assume the load address is a . If a is present in the store buffer of the processor, then the load returns the value of the *youngest* store for a in the local store buffer. Otherwise, the load is executed in either of the following two ways (the choice is arbitrary):
 1. The load returns the monolithic memory value $m[a]$, and all values for a in the local invalidation buffer are removed.
 2. The load returns some value for a in the local invalidation buffer, and all values for a older than the load result are removed from the local invalidation buffer. (If there are multiple values for a in the invalidation buffer, the choice of which one to read is arbitrary).
- **WMM-St** (store execution)
Predicate: The next instruction of a processor is a store.
Action: Assume the store address is a and the store value is v . The processor inserts the store $\langle a, v \rangle$ into its store buffer, and removes all values for a from its invalidation buffer.
- **WMM-Com** (Commit execution)
Predicate: The next instruction of a processor is a Commit and the store buffer of the processor is empty.
Action: The Commit fence is executed simply as a NOP.
- **WMM-Rec** (execution of a Reconcile fence)
Predicate: The next instruction of a processor is a Reconcile.
Action: All values in the invalidation buffer of the processor are removed.
- **WMM-DeqSb** (dequeue store buffer)
Predicate: The store buffer of a processor is not empty.
Action: Assume the value of the *oldest* store for *some* address a in the store buffer is v . First, the stale $\langle \text{address}, \text{value} \rangle$ pair $\langle a, m[a] \rangle$ is inserted to the invalidation buffer of every other processor whose store buffer does not contain a . Then this store is removed from the store buffer, and the monolithic memory $m[a]$ is updated to v .

Figure 12: Operations of the WMM Abstract Machine

processor executes or observes a store, values staler than the store are removed from the local invalidation buffer.

In order to enforce ordering in accessing the memory and the store buffer, we introduce the Commit fence. When a processor executes a Commit fence, it gets blocked unless its store buffer is empty. Similarly, we introduce the Reconcile fence. When a processor executes a Reconcile fence, the local invalidation buffer is immediately cleared. This prevents younger loads from reading the stale values.

Mapping C++11 Primitives to WMM

The Reconcile and Commit fences in WMM can be considered as acquire and release operations. Besides, the instruction sequence “Commit;Reconcile” forms a full fence. Figure 11 shows the mapping of C++11 primitives to WMM instructions and fences. Thanks to the clear definition of WMM, we have proven that the mapping is sound with respect to the C++11 language model.

Formal Analysis of Resilience Properties

While WMM is not the weakest memory model possible, e.g., see GAM [70], it is the most resilient to likely microarchitectural evolution paths. The weakest models require certain hard-to-understand constraints to make them programmable, and such constraints are brittle in the face of hardware changes.

Most memory models (e.g., ARM, POWER, RMO, Alpha, etc.) are defined as a set of axioms that check if a program behavior is legal or not. The major part of the axiomatic definition is to define how to compute the so-called *preserved program order* (PPO) for a given program behavior. Different memory models differ in the ways of computing PPO, but they share the same set of axioms as we have shown in [70]. Intuitively, PPO specifies in which cases the processor cannot reorder two instructions. If PPO contains less ordering cases, then the memory model is weaker. We have constructed the weakest PPO definition which is just enough to maintain single-threaded semantics and multi-threaded programmability [70].

This weakest PPO does not keep the load-to-store ordering in the general cases. However, if it allows load-store re-orderings in all cases, then the definition will permit the out-of-thin-air (OOTA) behaviors which will make any program analysis infeasible [13]. To forbid the OOTA behaviors, the PPO definition has to include some notions of dependency orderings. These dependency orderings are affected by hardware changes, because future processors with more aggressive speculation techniques (e.g., value predictions) may violate some dependency orderings. This explains why the weakest model is not resilient. Based on these observations, in the definition of WMM, we forbid load-store re-orderings to avoid OOTA behaviors. We have also shown in [69] that forbidding load-store re-orderings has little impact on performance.

3.2.4 Task 2.4: Integrated Validation of Control Theoretic Framework with Machine-Learning Discovery and Assurance Monitoring

3.2.4.1 Combining Learning and Control

A major goal of this project is to combine learned models with control systems. Intuitively, combining learned models of complex computer systems (including applications and the underlying hardware and system software) with control-theoretic resource management should produce predictable behavior (and long-lived software systems) in complex, dynamic environments.

We consider that both application and system have multiple configurations that can be altered at runtime. Application configurability is a direct result of writing applications in our FAST

language. Applications are made configurable by declaring *knobs* to indicate that a particular variable is dynamically alterable. Systems are already quite configurable and we assume common system knobs, such as core allocation or variable clock speed. Thus, considering all system and application knobs creates a huge space of possible configurations that can be dynamically altered at runtime. We would like to use control theory to provide guarantees that the application and system will achieve their goals, while using machine learning to produce estimates of the complicated tradeoff space.

To derive the benefit of both learning and control, however, requires addressing two major challenges:

- Dividing configuration management into sub-problems that suit learning and control’s different strengths.
- Defining abstractions that efficiently combine sub-problem solutions, while maintaining control’s formal guarantees.

We address the first challenge by splitting dynamic configuration into two sub-tasks. The first is learning relative behavior—speedup, for example, instead of absolute performance—so that all unpredictable external interference is viewed as a change to a *baseline* and the relative behavior is independent of these changes. Learning is well-suited to modeling relative behavior as a function of application and system configuration and finding Pareto-optimal tradeoffs. The second sub-task is controlling behavior dynamically based on the difference between measured and desired intent. Once the learner has found Pareto-optimal tradeoffs the problem is convex and well-suited to adaptive control solutions which guarantee the required intent even in dynamic environments.

The intuition is that computer system complexity creates local optima, where control solutions can get stuck; but learning finds true optimal tradeoffs—“convexifying”—the problem, allowing control techniques to handle dynamics while providing globally optimal outcomes. Even if the measured behavior is substantially changed due to external disturbances, the assurance monitor watches the controller and accounts for these dynamics.

We address the second challenge by defining an interface between learning and control that maintains control’s formal guarantees. This interface consists of two parts. The first is a lookup table that maps available configurable parameters (the knobs) to changes in behavior (observable metrics, or *measures* in FAST). The second part of the interface is the learned variance. Knowing this value, the controller can adjust itself to maintain formal guarantees even though the speedup is modeled by a noisy learning mechanism at runtime, rather than directly measured offline—as it would be in traditional control design. The key here is that given the noisy learning mechanism, the control guarantees become *probabilistic* instead of absolute. Hence the term ϵ -approximate control system—depending on the needs the Proteus controller can come arbitrarily close to the desired behavior of a traditional control system.

Thus, Proteus has developed a general methodology where an abstract control system is customized at runtime by a learning mechanism to meet requirements optimally. Unlike previous work on control systems that required numerous user-specified models and parameters, Proteus’ learner tunes the control parameters automatically; i.e. *it requires no user-level inputs other than the specified intent.*

This approach to using learning to customize control systems at runtime, ensuring application behavior both formally and empirically, is described in [35]. Note that this work received the

IEEE Micro Top Picks Honorable Mention for 2018. For additional technical detail about the empirical validation of the integrated system, see Appendix 5.2.3 and Section 5 of [65]. The way in which this approach is integrated in the FAST language is also described in [65].

3.2.4.2 Improving Learning for Computer Systems Discovery

Computer systems expose a wide variety of resources—including, but not limited to heterogeneous core types, multiple sockets, configurable memory hierarchies, adjustable clock speeds—and these resources have complex, non-linear effects on performance and energy. For example, on little cores clock speed might uniformly increase performance, while on big cores high clock speeds might induce thermal throttling, causing performance to decline. These types of resource interactions create local optima and make it difficult (or impossible) for gradient-based optimization and other heuristics to find a true optimal resource allocation. Indeed, several studies show that the increasing variety and complexity of configurable resources has rendered venerable heuristics ineffective.

Proteus uses machine learning techniques to increase system robustness by replacing resource management heuristics. Machine learning can model resources' complicated, non-linear interactions to avoid local optima and deliver a true optimal solution. Indeed, as systems complexity has grown researchers have proposed a variety of machine learning methods for system resource management. While this prior work shows that machine learning is effective for modeling complicated tradeoffs, there are several challenges that must be addressed to continue improving learning for computer system resource management, including:

- *Scarce Data:* To increase learning *accuracy*—*i.e.*, the learned model's ability to predict ground truth for some unseen application and system configuration—a robust set of training data is required. Collecting this training data is expensive: it requires observing a benchmark set in many different configurations, during which the machine to be modeled is not doing useful work. Additionally, the training benchmarks must exhibit a wide range of behavior, so that they can make accurate predictions for previously unseen applications.
- *Asymmetric Benefits:* Most learning problems require equal accuracy for all inputs. Furthermore, achieving better results for one input class represents a *biased* learner, a condition to be avoided, in general. In learning for systems, however, not all resource configurations are useful, in the sense of representing optimal tradeoffs (*e.g.* between performance and power).¹ Ideally, a computer system would only use configurations on the optimal frontier and ignore all non-optimal configurations; *i.e.*, unlike general learning problems, biasing the learner towards configurations representing optimal tradeoffs is beneficial. The challenge is that we do not know which configurations are optimal to begin.

We address these two challenges by presenting two techniques that improve a variety of learning methods for computer system management. First, we propose a novel *generative model* that addresses the scarce data challenge by generating training data that improves learning accuracy.

¹The set of optimal tradeoffs could either be Pareto-optimal, or more strictly, the set of inputs on the lower (or upper) convex hull of the tradeoff space, depending on the specific problem formulation. We simply use the term *optimal* tradeoffs as the ideas in this paper are common to both cases.

The key insight for the generative model is determining how to generate data that is sufficiently different from the training set, but still realistic enough to predict unseen behavior. Second, we propose *multi-phase sampling* to address asymmetric benefits by splitting sampling into two phases: first, separating the optimal configurations from the rest and second, improving the prediction accuracy of the optimal points.

We test these techniques by implementing them for both an ARM big.LITTLE mobile and an Intel x86 server. We use five different published learning systems to both predict performance and power consumption for unseen benchmark applications and to schedule resources to meet application latency requirements with minimal energy. We compare the results of these published learners to the same learners augmented with our proposed generative model and multi-phase sampling. Our results show the following:

- The generative model improves predictions for all learners on both the mobile and server systems. The average increase in prediction accuracy is 8 percentage points.
- Multi-phase sampling improves energy savings on both the mobile and server systems. On average—across all learners and systems—this technique produces energy that is 26% closer to optimal than the published learners for system resource allocation and energy management.

Additionally, our data support the following observations:

- While increasing learning accuracy generally reduces energy, *there is a point of diminishing returns*.
- Thus, even though the generative model improves even the best prior learner’s accuracy, *the continued accuracy improvement does not reduce energy*.
- Because multi-phase sampling biases the learner towards optimal configurations it *reduces overall accuracy, yet significantly improving energy savings*.

This study is strong evidence that after achieving a certain level of accuracy, it is no longer profitable for systems researchers to improve learning systems *without accounting for the structure—i.e., the geometry of optimal tradeoffs for the systems problem to be solved*. In our example of meeting latency requirements with minimal energy, the learners only need to produce accurate results for the configurations on the frontier of optimal performance and power tradeoffs. Once we have separated the optimal configurations, further accuracy improvements provide no additional energy savings and simply waste resources learning for no advantage. This problem is exacerbated because for any one given application most points are not on the optimal frontier. Thus, optimizing for learning accuracy improves predictions for points that are not practically useful.

More information on this study and methods for improving systems outcomes can be found in [18].

3.2.5 Task 2.6: Assurance Monitor to Trigger Transitions from Online to Offline Learning Methods

An assurance monitor is a system that observes the state trajectory of another system, predicts its future trajectory, and monitors the probability with which it will violate a property. Depend-

ing on the probability, it then triggers one of a set of actions, to bring the probability back to safe levels.

During Phase II, we developed a framework for assurance monitoring as a pattern language for event-based systems. In this framework, an assurance monitor consists of several collaborating components that:

1. Estimate the current state distribution.
2. Fuse this with control and sensor signals to predict a distribution of future state trajectories.
3. Check for violation of assurance properties over this distribution to trigger one or more remedial actions, to take the system back to a safe state. These actions can include using more of the available resources to increase the accuracy of the state and/or trajectory distributions, for example by activating additional sensors, or to compute a new and safer control plan.

This framework was implemented in Python atop of the IoTPy framework for event-based systems [28]. It was further evaluated using two example applications:

1. Autonomous drone navigation, based on the CaPSuLe image matching algorithm [37] that underlies the Phase II/III Challenge Problem 3 (Section 4.1.1).
2. A distributed earthquake prediction system [20] that uses the assurance monitor to manage time accuracy, trading precision against the cost of eliminating clock drift by consulting a network time server.

The source code is available on the MIT GitHub [29].

3.3 Technical Area: Novel Control-Theoretic Analytics Supported by Machine Learning

3.3.1 Task 3.1: Offline Machine-Learning Models and Model Coherence with Online Learning

In conventional computing, programs are usually treated as implementations of mathematical functions, so there is a precise output that must be computed for a given input. In many problem domains, it is sufficient to produce some approximation of this output; for example, when rendering a scene in graphics, it is acceptable to take computational short-cuts if human beings cannot tell the difference in the rendered scene. In this work, we focus on a class of approximate programs that we call *tunable approximate* programs. Intuitively, these programs have one or more knobs or parameters that can be changed to vary the fidelity of the produced output. These knobs might control the number of iterations performed by a loop [14, 45], determine the precision with which floating-point computations are performed [47, 49], or switch between precise and approximate hardware [19]; for the purposes of this work, the source of approximation does not matter so long as the fidelity of the output is changed by adjusting the knobs.

Roughly speaking, given a permissible error for the output, we want to set the knobs to minimize computational costs, such as running time or energy, while meeting the error constraint. This work describes a solution to the proactive control problem for non-streaming programs that consist of components controlled by one or more knobs and in which the error and cost behaviors are substantially different for different inputs. Our approach is to treat the control problem as a constrained optimization problem in which an objective function such as energy is minimized, subject to constraints such as a lower bound on the acceptable output quality. The major challenge is that this formulation requires us to know the objective and constraint functions, but in general these are complex functions that we do not know and cannot write down in closed form. We deal with this by modeling these functions using machine learning techniques. The resulting control system [53], which is an example of open-loop control [9], is fairly successful in controlling approximation in a principled way in complex applications from several domains including machine learning, image processing and graph analytics.

3.3.1.1 Problem Formulation

We describe the formulation of the proactive control problem we use in this work, justifying it by describing other reasonable formulations and explaining why we do not use them. To keep notation simple, we consider a program that can be controlled with two knobs K_1 and K_2 that take values from finite sets κ_1 and κ_2 respectively. We write $K_1 : \kappa_1$ and $K_2 : \kappa_2$ to denote this, and use k_1 and k_2 to denote particular settings of these knobs. The formulation generalizes to programs with an arbitrary number of knobs in an obvious way.

It is convenient to define the following functions.

- **Output:** In general, the output value of the tunable program is a function of the input value i , and knob settings k_1 and k_2 . Let $f(i, k_1, k_2)$ be this function.
- **Error/quality degradation:** Let $f_e(i, k_1, k_2)$ be the magnitude of the output error or quality degradation for input i and knob settings k_1 and k_2 .
- **Cost:** Let $f_c(i, k_1, k_2)$ be the cost of computing the output for input i with knob settings k_1 and k_2 . This can be the running time, energy or other execution metric to be optimized.

We formulate the control problem as an optimization problem in which the error is bounded for the particular input of interest. This optimization problem is difficult to solve, so we formulate a different problem in which the expected error over all inputs is less than the given error bound, with some probability. This gives the implementation flexibility in finding low-cost solutions.

One way to formulate the control problem informally is the following: given an input value and a bound on the output error, find knob settings that (i) meet the error bound and (ii) minimize the cost. This can be formulated as the following constrained optimization problem.

Problem Formulation 1. *Given:*

- *a program with knobs $K_1: \kappa_1$ and $K_2: \kappa_2$, and*
- *a set of possible inputs I .*

For input $i \in I$ and error bound $\epsilon > 0$, find $k_1 \in \kappa_1, k_2 \in \kappa_2$ such that

- *$f_c(i, k_1, k_2)$ is minimized*
- *$f_e(i, k_1, k_2) \leq \epsilon$*

In the literature, the constraint $f_e(i, k_1, k_2) \leq \epsilon$ is said to define the *feasible region*, and values of (k_1, k_2) that satisfy this constraint for a given input are said to lie within the feasible region for that input. The function $f_c(i, k_1, k_2)$ is the *objective function*, and a solution to the optimization problem is a point that lies within the feasible region and minimizes the objective function.

For most tunable programs, this is a very complex optimization problem since the Pareto-optimal knob settings vary greatly for different inputs [53]. To get a sense of this complexity, consider the GEM benchmark, a graph partitioner for social network graphs [62]. We conducted experiments involving running GEM with a variety of inputs and different knob settings, and measuring the cost (running time of the program) and error of the output of the resulting programs. We observed that even for a single input graph, there are many knob combinations that produce the same output error, and that these combinations have widely different costs.

For a given input graph and output error, we are interested in minimizing cost. Since Pareto-optimal frontiers are very different for different inputs, it is difficult to produce the Pareto-optimal knob settings for a given input and output error without exploring much of the space of knob settings for a given input, which is intractable for non-trivial systems.

One way to simplify the control problem is to require only that the *expected* output error over all inputs be less than some specified bound ϵ . Since some inputs may be more likely to be presented to the system than others, each input can be associated with a probability that is the likelihood that input is presented to the system. This lets us give more weight to more likely inputs, as is done in Valiant’s probably approximately correct (PAC) theory of machine learning [59]. Since the cost function is still a function of the actual input, knob settings for a given value of ϵ will be different in general for different inputs, but the output error will be within the given error bounds only in an average sense. In our approach, we consider a variation of this optimization problem, inspired by Valiant’s work [59], in which we are also given a probability π with which the error bound must be met. Intuitively, values of π less than 1 give the control system a degree of slack in meeting the error constraint, permitting the system to find lower cost solutions. This control problem can be formulated as an optimization problem as follows.

Problem Formulation 2. *Given:*

- *a program with knobs $K_1 : \kappa_1$ and $K_2 : \kappa_2$,*
- *a set of possible inputs I , and*
- *a probability function p such that for any $i \in I$, $p(i)$ is the probability of getting input i .*

For an input $i \in I$, error bound $\epsilon > 0$, and a probability $1 \geq \pi > 0$ with which this error bound must be met, find $k_1 \in \kappa_1$, $k_2 \in \kappa_2$ such that

- *$f_c(i, k_1, k_2)$ is minimized*
- $$\sum_{(j \in I) \wedge (f_e(j, k_1, k_2) \leq \epsilon)} p(j) \geq \pi$$

If the term $\sum_{(j \in I) \wedge (f_e(j, k_1, k_2) \leq \epsilon)} p(j)$ (denoted by $P_e(\epsilon, k_1, k_2)$) is greater or equal to π , then (k_1, k_2)

is in the feasible region for error bound ϵ . For future reference, we call this the *fitness* of knob setting (k_1, k_2) for error ϵ ; intuitively, the greater the fitness of a knob setting, the more likely it is that it satisfies the error bound for the given ensemble of inputs. In the rest of this section, we refer to Problem Formulation 2 as the “control problem.”

3.3.1.2 Proactive Control for Approximate Programs

For the complex applications we are interested in, the error function $f_e(i, k_1, k_2)$ and the cost function $f_c(i, k_1, k_2)$ are non-linear functions of the inputs, and it is difficult if not impossible to derive closed-form expressions for them. Therefore, we use machine learning techniques to build surrogates for these functions offline, using a suitable collection of training inputs. Figure 13 is an overview of the control system, which we call *Capri*. For a given program, the system must be provided with a set of training inputs, and metrics for the error/quality of the output and the cost. The offline portion of the system runs the program on these inputs using a variety of knob settings, and learns the functions f_e and f_c . These models are inputs to the controller in the online portion of the system; given an input and values of ϵ and π , the controller solves the control problem to estimate optimal knob settings. In the following, we describe the major modules of the Capri control system.

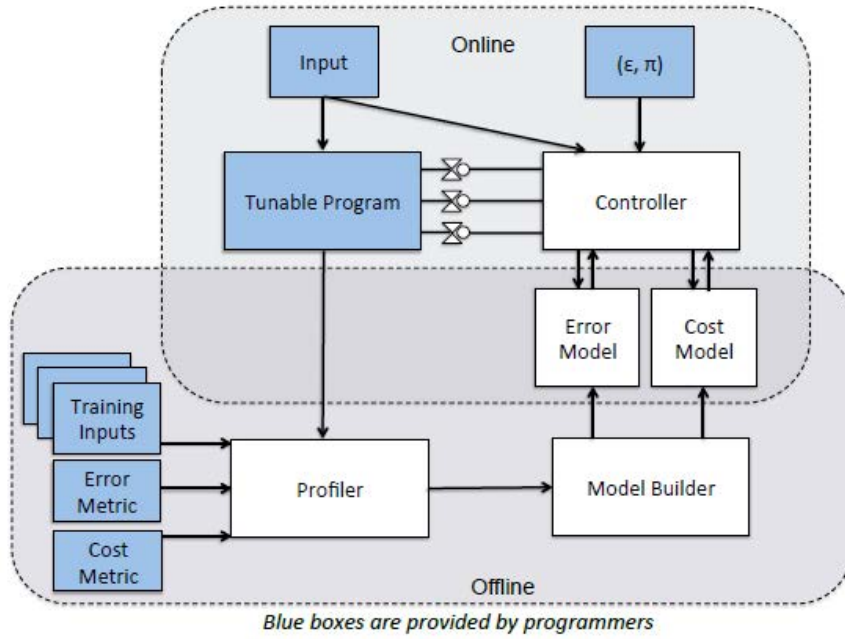


Figure 13: Overview of the Capri Control System

Error Model

The error model is a surrogate for the fitness function $P_e(\epsilon, k_1, k_2)$ and is used to determine whether a knob setting is in the feasible region. Intuitively, a knob setting is in the feasible region if the inputs for which the error is between 0 and ϵ have a combined probability mass greater than or equal to π .

In our published work on Capri [53], we used Bayesian networks [40] to model error. Our proposed Bayesian network model was simple, in which each of the knobs and the error is modeled as a random variable and the output error depends on all of the knobs. Such a Bayesian error model has the virtue of being simple, but it does not scale since the size of the conditional probability distribution tables increases exponentially with the number of knobs. In our studies, we have found that the probability function changes quite slowly as the knob settings are changed. Therefore, we might be able to usefully approximate the probability table by partitioning the knob space into subspaces and using a simple linear model, such as M5 [42]), within each subspace.

Cost Model

The cost model is the surrogate for $f_c(i, k_1, k_2)$. We model both the running time and total energy. For most algorithms, the running time can vary substantially for different inputs; after all, even for simple algorithms like matrix multiplication, the running time is a function of the input size. For complex irregular algorithms like the ones considered in this proposal, running time will depend not only the input size but also on other features of the input. For example, the running time of a graph clustering algorithm is affected by the number of vertices and edges in the graph as well as the number of clusters. Therefore, the running time is usually a complex function of input *features* and knob settings. Our system currently requires the user to specify what these features are

Controller

The control algorithm must search the space of knob settings to find optimal knob settings, using the error and cost models as surrogates for f_e and f_c respectively. Our system is implemented so that new search strategies can be incorporated seamlessly. This lets us evaluate model accuracy separately from search accuracy. We evaluated two search algorithms: *exhaustive search* and *Precimonious search* [47]. If the error and cost models are not expensive to evaluate and each knob has a finite number of settings, we can use exhaustive search. We sweep over the entire space of knob settings, and for each knob setting, use the error model to determine if that knob setting is in the feasible region. The cost model is then used to find a minimal cost point in the feasible region. In a large search space, heuristics-based search is an effective way to trade-off search cost for quality of the result. Precimonious search, which is based on the delta-debugging algorithm [22], is one such strategy. The algorithm starts with all knobs set to the highest values, and attempts to lower these settings iteratively. Precimonious can quickly prune the search space but the solution it finds may be a local minimum. Other search strategies can be implemented easily within Capri.

3.3.2 Task 3.3: Definition of the FAST Language and Implementation of Selected Features

Applications that adapt to their environment have been proposed [10, 23, 25, 27, 54] as an alternative to costly application maintenance and defensive programming. Beyond what judicious up-front implementation can offer, such applications have the ability to adapt while the application is running. Constructing adaptive applications has thus far required the developer to augment their code with considerable boiler-plate code, and to program their application in C or C++. While highly efficient, these languages lack some of the properties of modern languages that enhance productivity and facilitate correctness of implementation.

The rest of this section elaborates on the intent-driven programming model, which address the first problem outlined above. The language FAST, developed as part of this project, supports this programming model. We address the second problem by making FAST an extension of the Swift programming language that comes with strong memory- and type safety guarantees, as well as a flexible syntax amenable to embedding languages such as FAST.

3.3.2.1 Intent-Driven Programming Model

Given the growing importance of energy-efficient computation in both mobile computing and high-performance supercomputing, inexact computing has gained prominence as a strategy for reducing the energy consumption while possibly increasing the performance of computing applications. Much of this work has concentrated on potential energy gains from inexact hardware with reduced voltages and concomitant intermittent errors. In such scenarios, the behavior of hardware becomes probabilistic rather than deterministic, yielding statistical rather than deterministically defined results. Palem, Roy, and others have observed that major energy savings (measured in orders of magnitude) are possible if hardware accuracy is compromised. The merit of this approach to developing computing applications has been demonstrated by Palem in building embedded systems with inexact hardware components and non-deterministic behav-

ior that is nearly indistinguishable from much higher energy, deterministic counterparts. More recently, Ceze and Grossman have developed the EnerJ programming language based on Java for expressing such computations with non-deterministic results in high-level form. In EnerJ, inexact computations are partitioned from exact computations by the type system to enforce the invariant that results of conventional (exact) type may not depend on the results of inexact subcomputations.

A related line of research has focused on coping with uncertainty in program inputs such as GPS data. A GPS measurement can be conceptualized as a probability distribution around the measured value. To cope with uncertainty in computing applications, McKinley and Mytowicz at Microsoft Research have proposed incorporating *uncertain* data types in programming languages and using statistical analysis such as Bayesian inference in compilation to support conditional operators with specified certainty. Programmers with little knowledge of statistics can use these abstractions to produce statistically correct applications.

In contrast, intent-driven programming focuses entirely on “exact” computation performed using common off-the-shelf (COTS) hardware. Although floating point numerical computation is technically exact and reproducible, it is typically analyzed as a non-deterministic computational medium where the results of computations over the real numbers are approximations of the corresponding exact answers (using real arithmetic) smeared by the effects of potential “round-off” error in every primitive numerical operation (addition, subtraction, multiplication, division). Hence the distinctions between exact and inexact hardware are not nearly as crisp in practice as a theoretical analysis might suggest. In contrast to EnerJ, the answers computed by applications may depend on approximate subcomputations performed using lower precision floating point arithmetic or less accurate iterative algorithms when appropriate. For example, low-precision floating point arithmetic may be used in exploratory phases of branch-and-bound algorithms to rule out unpromising portions of the search space. A precise (and computationally expensive) version of an intent-driven program is typically used as the reference for determining the quality of the answers generated by more energy efficient versions of the same program.

The critical novelty in intent-driven programming is the *dynamic* adjustment of program semantics to achieve specified trade-offs between quality (typically accuracy) and computational cost (typically energy usage). This approach has been pioneered by Hank Hoffman and his students in the context of existing applications written in C. The language construct dubbed *optimize* in our programming model is implemented by an application specific library and supporting offline profiling system called POET. Our intent-driven programming model provides a linguistic framework that generalizes Hoffman’s work to a wide range of applications and automates the appropriate profiling process. The profiling problem has been studied in more depth by Fussell and Pingali who interpret the problem in the domain of machine learning. Based on profiling data, their system infers cost and quality functions (over the tunable program parameters of the application) that are stored in a database for subsequent use at runtime. In contrast to Hoffman’s POET system, the Fussell/Pingali Capri system is less dynamic. It can dynamically adjust knobs in a streaming program to each input using the machine-learned cost and quality functions but it does not use any feedback from the results of earlier computations. Our programming model supports both feedback-based and feedback-oblivious forms of optimization. The former is applied to the optimization of streaming computations and the latter to

individual computations that do not appear within a stream. The types of the arguments passed to the optimize construct reveal which case pertains.

3.3.2.2 FAST Language Definition

The FAST language, an instance of the intent-driven programming model, is described in [65], which covers the user interface of the language, the runtime, and their role in the evaluation of the Proteus system.

3.3.2.3 METAFast Language Extension

Following the completion of the original Proteus project, we developed the METAFast language generalizing the semantics of FAST, an extension of the Swift programming language supporting intent-based control and optimization. The programming interface for METAFast is identical to that of FAST; METAFast differs from FAST in significantly relaxing run-time restrictions on the use of intent to control knob settings. In particular, any function in a METAFast program can dynamically load an intent (each intent is specified in a separate file), enabling a METAFast application to simultaneously adapt to multiple intents. Section 3 describes these differences in detail.

The primary deliverable for METAFast is a github repository ¹ consisting of two major parts:

- Package *Adapt*, which contains the library supporting METAFast, notably the generic *ControllableFunction* which can take any function and associate it with a corresponding intent-driven controller.
- Package *FlightTest* consisting of the METAFast code for our challenge problems (defined later).

3.3.2.4 Challenge Problems

To demonstrate the added capabilities of METAFast, we developed two challenge problem applications from the flight test domain, based on signal processing algorithms from the SIDPAC library [38].

Challenge Problem E1

The first challenge problem (E1) implements two tunable filters, executing simultaneously, to process a common input stream. The filters are:

- Recursive Discrete Fourier Transform (RDFT)
- Recursive Least Squares (RLS)

¹<https://github.com/darpa-brass/MetaFast>

Challenge Problem E2

The second challenge problem (E2) implements two tunable filters, executing in a pipeline, to process an input stream received by the first filter. The pipeline stages are:

- Recursive Discrete Fourier Transform
- Low-pass filter (LPF)

The tunable filters are controlled by intent files specifying the measure (computed quantity) to be optimized subject to a specified constraint.

3.3.2.5 Results and Accomplishments

Expressing Multiple Intents in METAFast

Like its FAST predecessor, METAFast separates intent files from the application code. In the application code, every function under control can load an intent at any time as long as the measure and knob declarations in the intent match the declarations in the function. The syntax of intent is exactly the same as in FAST. The only difference is that METAFast allows users to associate multiple functions with multiple intents and to call those functions anywhere in the program. METAFast controls the average cost of each function according to its intent.

In Figure 14, we show the intent files for E1.

1 knobs	1 knobs
2 period from [5 reference,10,15,20,25]	2 filterSize from [5 reference
3 successive from [0, 1 reference]	,6,7]
4 coreFrequency from [300 reference,1200]	3 successive from [0, 1
5	reference]
6 measures	4
7 latency: Double	5 measures
8 MSE : Double	6 latency: Double
9	7 RE : Double
10 intent RDFT	8
11 min(latency)	9 Intent RLS
12 such that MSE <= 200.0	10 min(latency)
	11 such that RE <= 20.0

(a) Intent specifications for RDFT.

(b) Intent specifications for RLS.

Figure 14: An Example of METAFast intent files.

Controlling with Multiple Intents

Here we show four experiments that run two functions associated with different intents under METAFast control. The first two experiments have the same function body with different intent goals while the latter two control a different function using different intents, corresponding to our challenge problem E1.

1. One RDFT with intent minimizing latency given $MSE \leq 100$ and another RDFT with intent minimizing latency given $MSE \leq 150$. The results are shown in Figure 15.
2. One RDFT with intent minimizing latency given $MSE \leq 100$ and another RDFT with intent minimizing latency given $MSE \leq 200$. The results are shown in Figure 16.

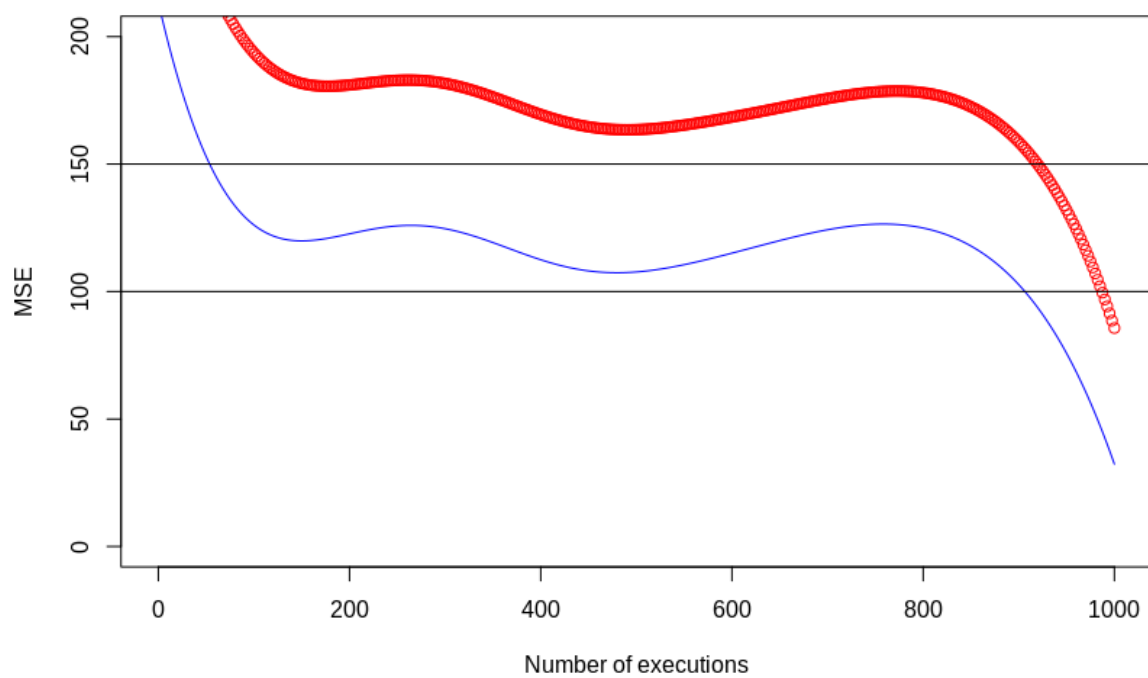


Figure 15: The results of running two RDFT functions with different intent goal ($MSE \leq 100$ (red) and $MSE \leq 150$ (blue) controlled by METAFast.

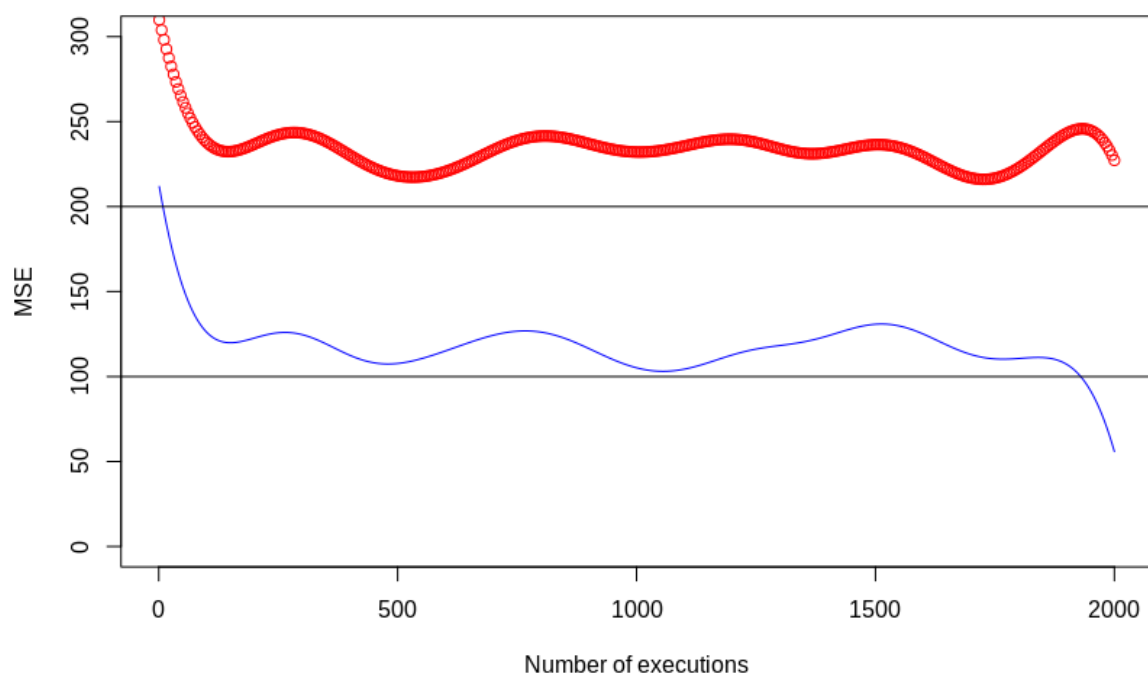


Figure 16: The results of running two RDFT functions with different intent goal ($MSE \leq 100$ (red) and $MSE \leq 200$ (blue) controlled by METAFast.

3. One RDFT with intent minimizing latency given $MSE \leq 100$ and one RLS with intent minimizing latency given $RE \leq 0.3$. The results are shown in Figure 17.
4. One RDFT with intent minimizing latency given $MSE \leq 200$ and one RLS with intent minimizing latency given $RE \leq 20$. The results are shown in Figure 18.

3.3.2.6 Hierarchical Intents and Control through Proteus

As in the last section, we show two additional experiments of running two functions, one calling another, associated with different intents under METAFast control. We call this scenario *hierarchical*, corresponding to our challenge problem E2. We have one invocation of RDFT with the intent minimizing latency given $MSE \leq 200$ and one invocation of LPF with the intent minimizing latency given $MSE \leq 200$. The results are shown in Figure 19. Note that the intent looks the same only because they share the same measure names. They are interpreted differently.

3.3.3 Task 3.4: Online Learning Algorithms Transferred to Analytics Team

This task's goal is to incorporate online learning models into the control system. This is an improvement over the base system Proteus used in Phase I, in which a single learned model was passed to the controller at system startup. In this new, online approach the controller can signal to the learner that it wants a new model and then incorporate that model as it executes.

3.3.3.1 Overview

Figure 20 shows the Proteus approach of splitting decision making into separate learning and control tasks and then composing these individual solutions. As the application executes, an adaptive control system allocates resources using a model built with the offline learning mechanisms developed in Year 1. While executing the controller records relevant metrics specified through the FAST language (e.g., latency, power, accuracy). The controller naturally predicts future values of any controller dimensions. If those values are not what it expects, then the recorded values are sent to an online learner, which builds a new model of the relevant metrics. The learner extracts those that are predicted to be Pareto-optimal and packages them in a data structure: the performance hash table (PHT).² The PHT and the estimated variance are sent to the controller, which sets its pole and selects an energy minimal resource configuration with formal guarantees of convergence to the desired behavior.

Figure 21 illustrates the asynchronous interaction between the learner and controller. The controller starts using a conservative, generic model produced by the offline learner. If the controller wants a new model at some point, it sends the learner the application's name and device type (message 1, Figure 21). The learner determines how many samples are needed for an accurate estimation and sends this number to the controller (message 2). The controller takes these samples and sends the latency and power of each measured configuration to the learner (message 3). The learner may require time to make predictions; so, the controller does not wait,

²Described in detail in the Year 1 report

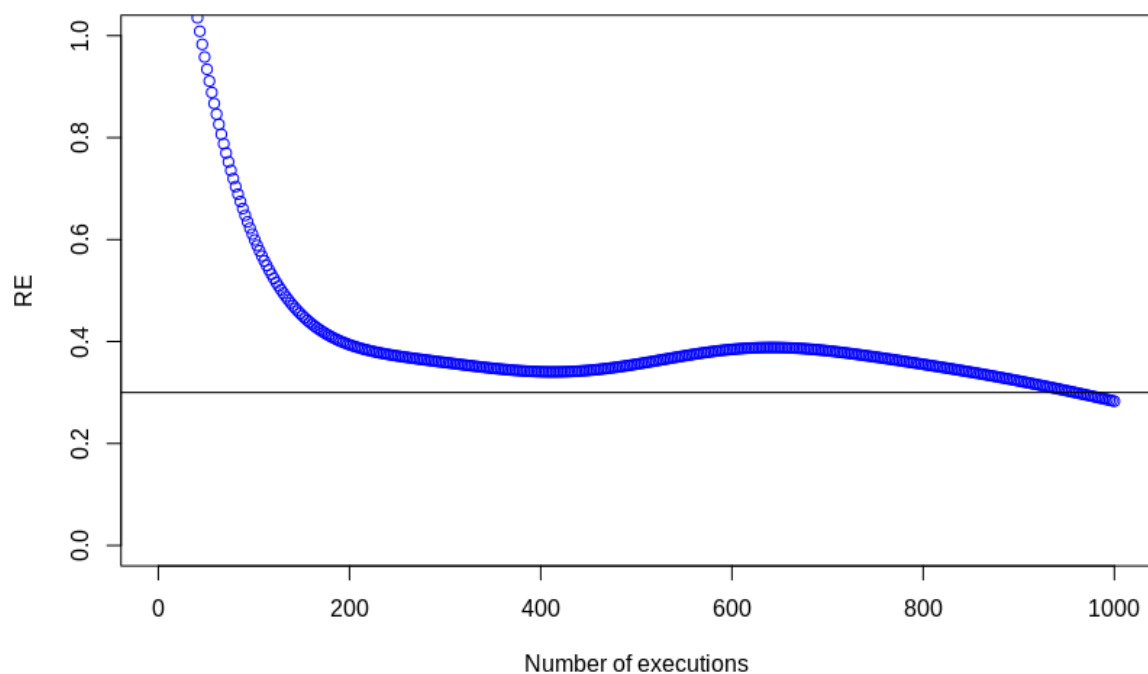
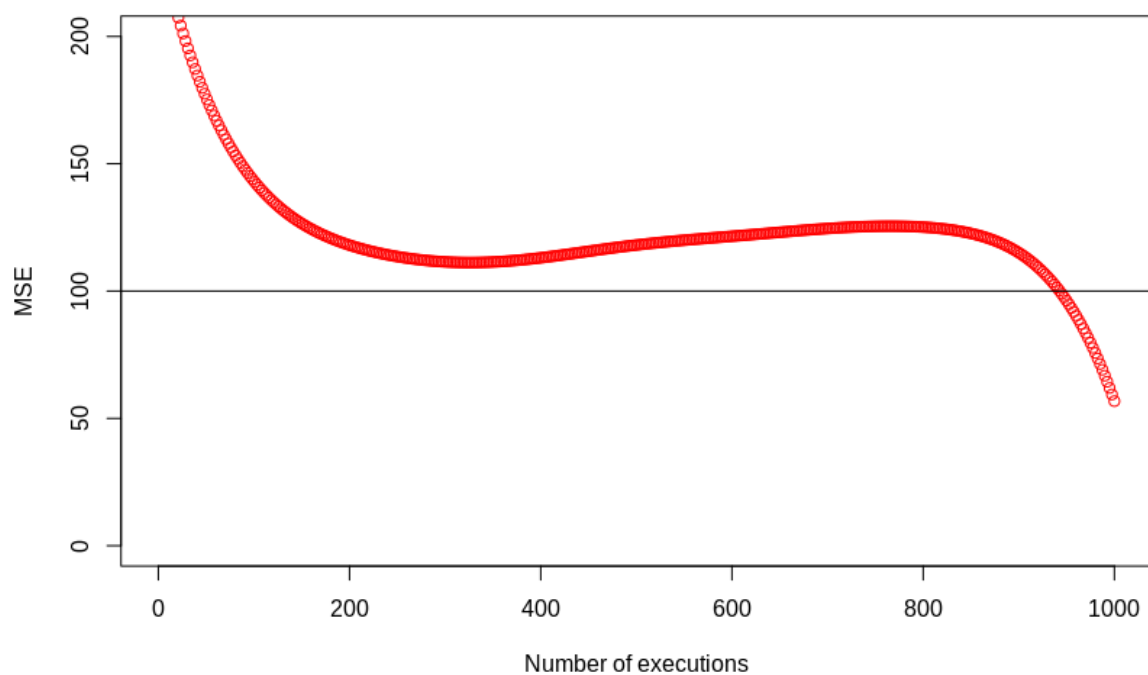


Figure 17: The results of running one RDFT functions with intent goal $MSE \leq 100$ (red) and one RLS with intent goal $RE \leq 0.3$ (blue) controlled by METAFast.

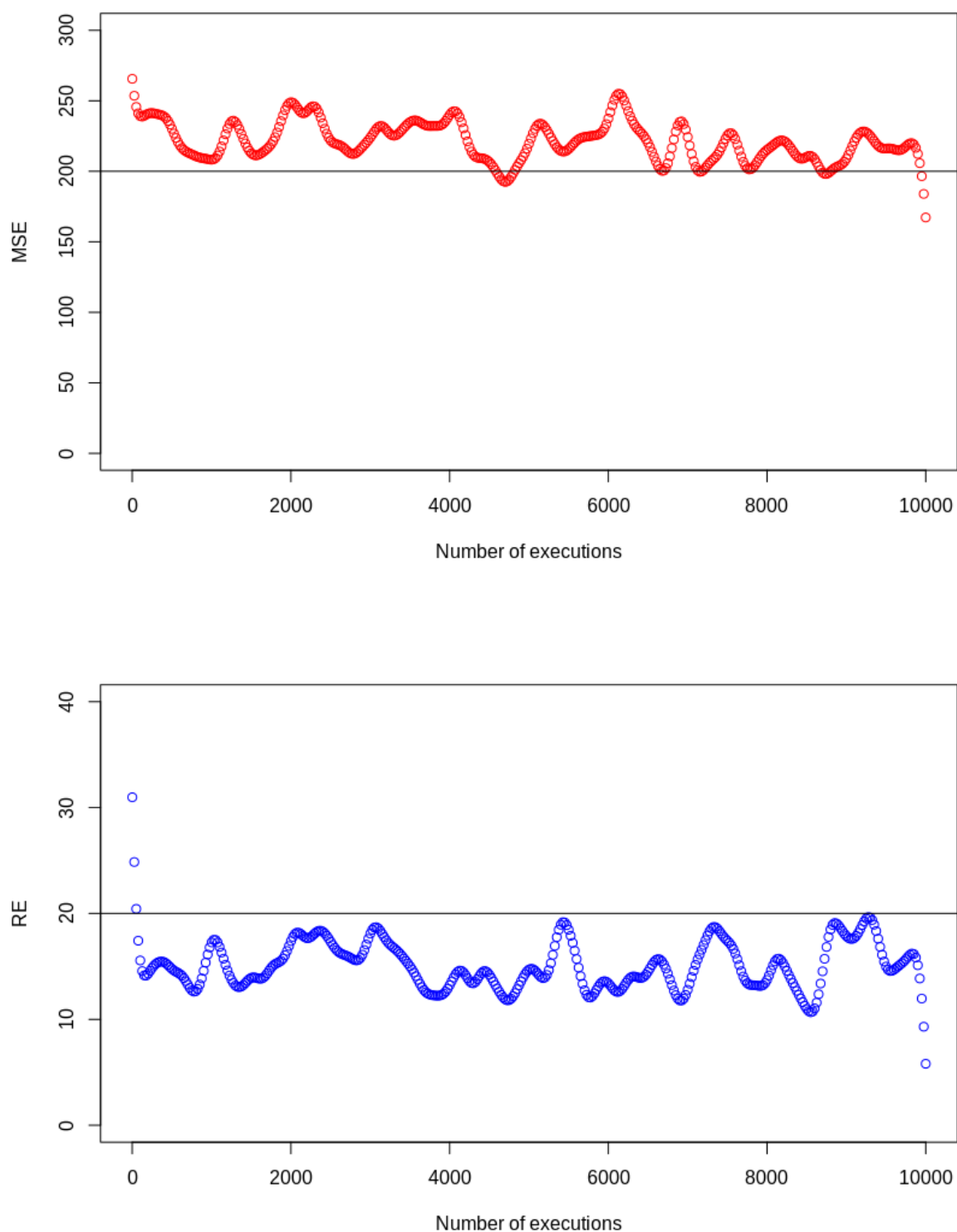


Figure 18: The results of running one invocation of RDFT functions with intent goal $MSE \leq 100$ (red) and one invocation of RLS with intent goal $RE \leq 0.3$ (blue) controlled by METAFast.

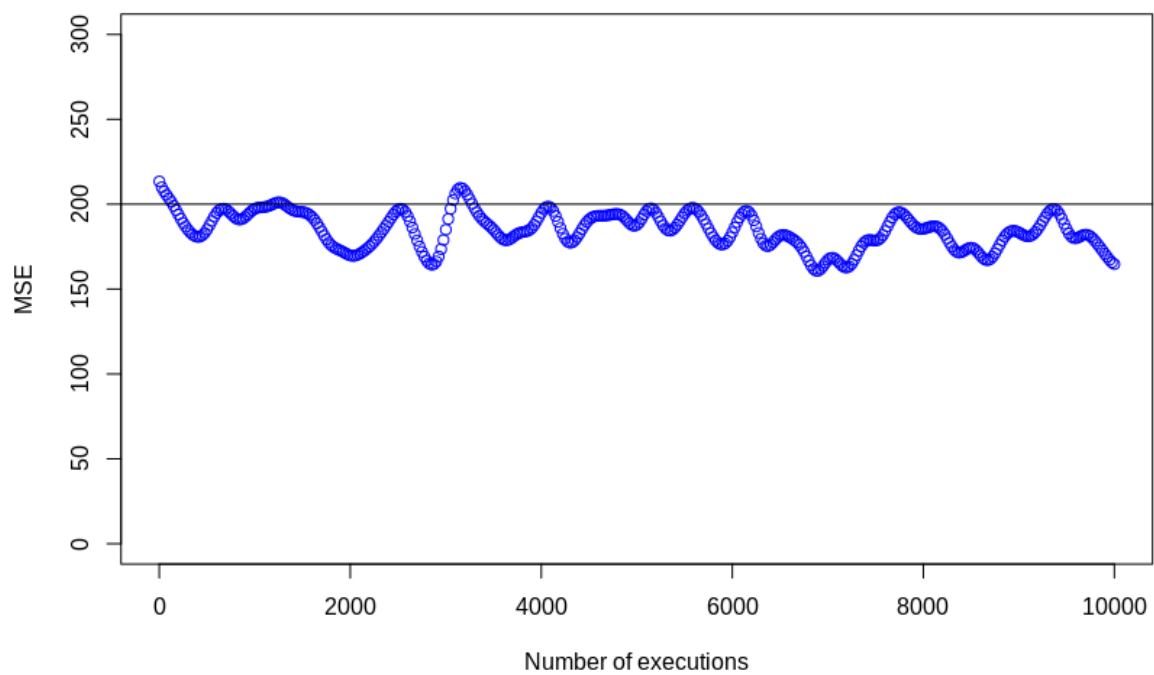
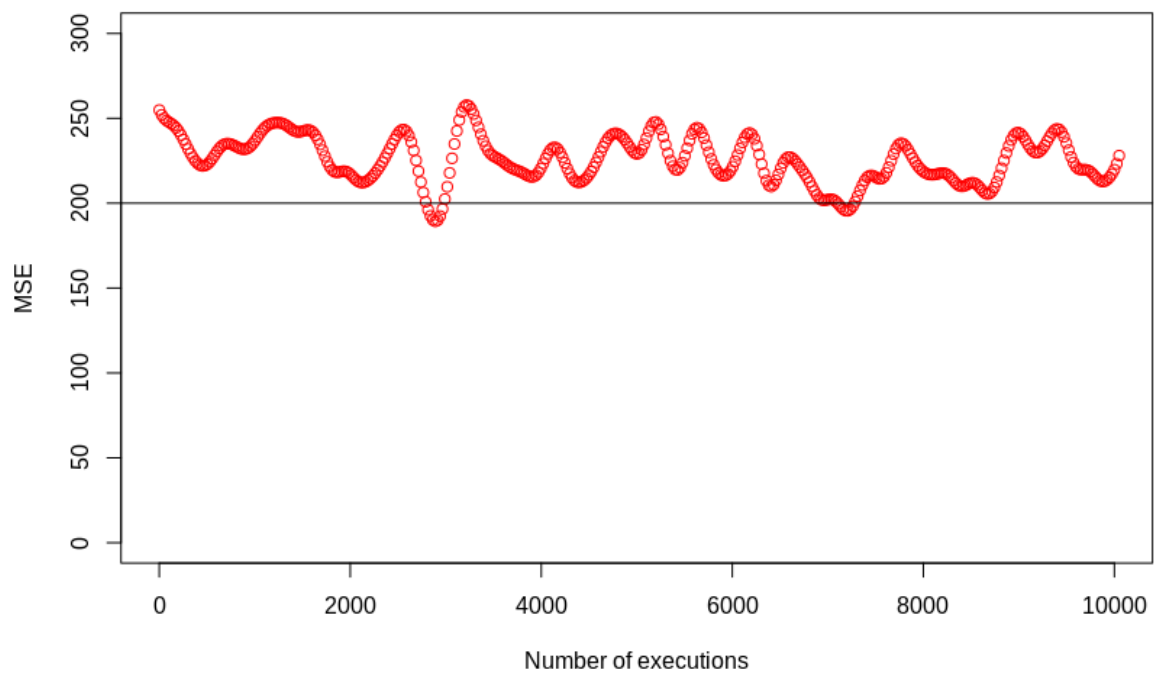


Figure 19: The results of running one invocation of RDFT functions with intent goal $MSE \leq 100$ (red) and its caller, an invocation of LPF with intent goal $MSE \leq 200$ (blue), both controlled by METAFast.

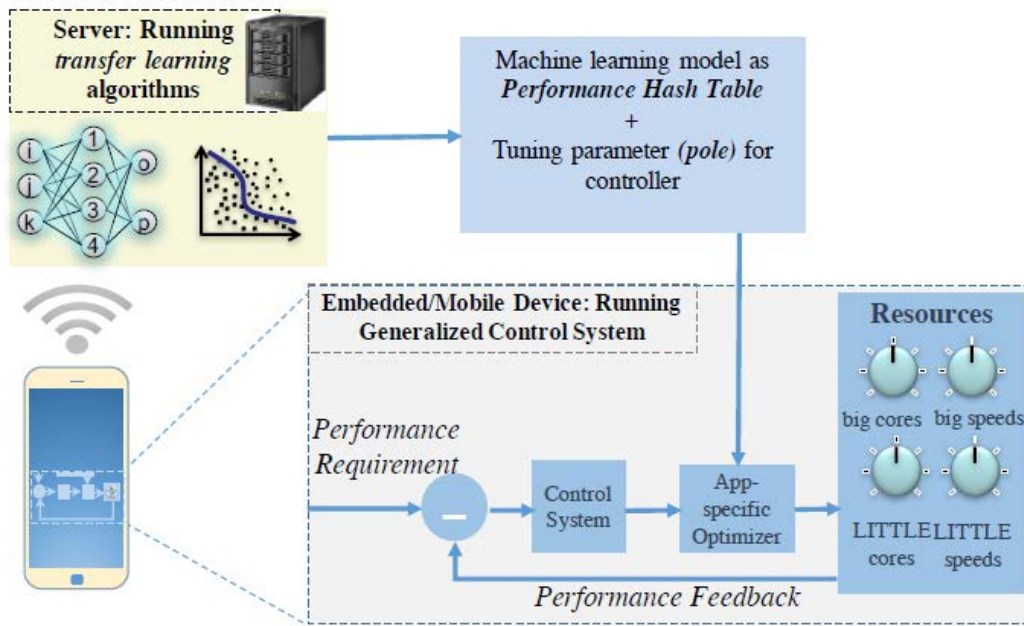
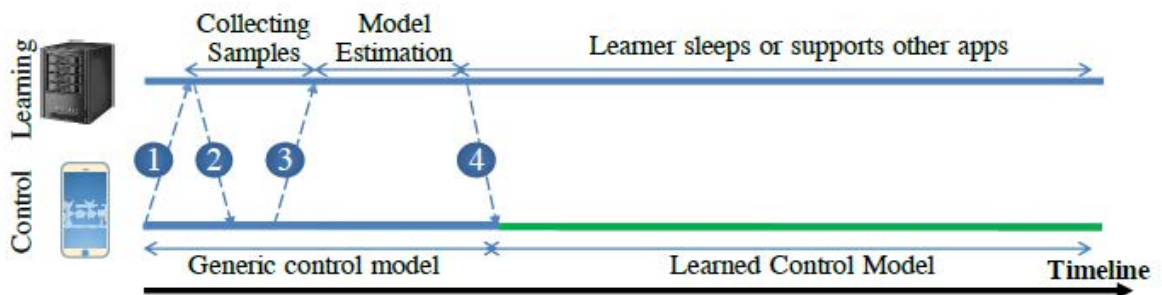


Figure 20: Schematic Diagram for Combining Learning and Control



- 1 An application starts and the controller begins with a generic model and queries the learner for the number of samples to take.
- 2 The learner responds with the number of samples needed, the controller continues.
- 3 The controller sends its samples back to the learner which asynchronously assembles a model.
- 4 The learner responds with a model customized for the application.

Figure 21: Online Communication of Learning and Control

but continues with its current model. Once the learner has a new model ready, it sends that data and the variance estimate to the controller (message 4), which uses the new learned model from then on.

Figure 21 shows several key points about the relationship between learning and control. First, the controller never waits for the learner: it uses the best available model until the learner produces a new, better update. Second, the controller does not continuously communicate with the learner—this interaction happens infrequently, as initiated by the controller. Third, if the learner crashed, the controller defaults to be a generic adaptive control system. If the learner crashed after sending its predictions, the controller does not need to know. Finally, the learner and controller have a clearly defined interface, so they can be run in separate processes or physically separate devices. In fact, the figure illustrates a learner running on a remote server while the controller operates on a mobile device. This separation may allow very expensive learning algorithms to be employed to manage small, energy-efficient mobile systems.

3.3.3.2 Accounting for Learned Models

One important lesson learned in Year 2 is that the control must account for errors in learned models. This fact is especially true for online learning methods that, by necessity, must be computationally cheaper than offline methods and therefore may sacrifice even a small amount of accuracy.

The controller's pole $\rho(t)$ is critical to providing control theoretic guarantees in the presence of learned—rather than directly measured—data. Therefore, the Proteus controller requires any learner estimate not only speedup and powerup, but also the variance σ . The Proteus controller uses this information to derive a lower bound for the pole which guarantees probabilistic convergence to the goal. Specifically, we prove that with probability 99.7% Proteus control converges to the desired latency if the pole is

$$[1 - \lfloor \max(\hat{s}) / (\min(\hat{s}) - 3\sigma) \rfloor_0]_0 \leq \rho(t) < 1,$$

where $\lfloor x \rfloor_0 = \max(x, 0)$ and $\hat{s}$ is the estimated speedup. Users who need higher confidence can set the scalar multiplier on σ higher; *e.g.* using 6 provides a 99.99966% probability of convergence.

Thus we provide a lower-bound on the value of $\rho(t)$ required for confidence that Proteus converges to the desired goal. This pole value only considers the controlled dimension, and not the optimality of the control. In practice, we find it better to use a higher pole based on the *uncertainty* between the controller's observed efficiency and that predicted by the learner. We follow prior work [57] in quantifying uncertainty as $\beta(t)$, and setting the pole based on this uncertainty:

$$\begin{aligned} \beta(t) &= \exp\left(-\left(\left|\frac{\bar{s}(t)}{\bar{p}(t)} - \frac{\hat{s}(t)}{\hat{p}(t)}\right|\right)/5\right) \\ \rho(t) &= \frac{1-\beta(t)}{1+\beta(t)} \end{aligned} \tag{7}$$

where $\bar{s}$ and $\bar{p}$ are the measured values of speedup and powerup and $\hat{s}$ and $\hat{p}$ are the estimated values from the learner. This measure of uncertainty captures tradeoffs in power and latency, but any tradeoff space managed by FAST can be substituted. We find that it is generally higher than the pole value given by our lower bound, so in practice Proteus sets the pole dynamically to be the higher of the two values.

We also note that the higher the uncertainty, the slower the system will respond to outside events. Generally this is a good property, we do not want our controller to make major changes when the model is not accurately predicting behavior. Too much uncertainty can, however, cause the system to react too slowly. Thus we are basing our decisions on when to request a new model on the uncertainty value. Right now we use the simple heuristic that if the pole produced by Eqn. 7 is higher than that produced by the required lower-bound, then it is time to request a new model. The intuition behind this heuristic is that this condition will only hold when the controller is seeing higher errors than the learner's own estimate of system variance. We believe this condition is a likely indicator that the underlying model is wrong.

3.3.3.3 Offline Learning

Prior work called *Capri* has shown that applications can be controlled using machine learning techniques [53] in an offline fashion. The Capri system works by profiling the applications that need to be controlled offline, and learns the application behavior with respect to metrics of interest using regression models. Given a set of goals and constraints that need to be met during application execution, Capri predicts the optimal knob configurations. As part of the Proteus project, we have reimplemented Capri to make the technique scale to more knobs, and have shown Capri's effectiveness for a range of complex applications [53, 55].

An important assumption in offline control techniques like Capri is that the models learned offline continue to represent the application and system behavior during execution. This is restrictive, since in reality, the behavior of the application may change substantially since the profiling phase in lieu of phases in the application, changes in the system dynamics, and environmental changes. This leaves techniques like Capri open to criticism—models learned by profiling are not updated after the training phase, and hence the subsequent control can be sub-optimal.

For online learning, the model built offline is updated with data measured during the actual run. Often, the output measured from the execution will differ from what the ML model predicts. This difference can stem from noise in execution, noise in measurement, and inaccuracies in the ML model estimation, *e.g.* reflecting a change in the input distribution. Therefore, a fundamental question here is “how to optimally fuse data from multiple sources (for *e.g.*, from the ML estimator and the measured output) which can be noisy”?

3.3.3.4 Kalman filtering for Proteus

Kalman filtering is a classic state estimation technique invented in 1960 by Rudolf E. Kálmán [30]. It is used in many areas including spacecraft navigation, motion planning in robotics, signal processing and wireless sensor networks [21, 39, 52, 56, 61, 63] because of its small computational and memory requirements, and its ability to extract useful information from noisy data. Recent works shows how Kalman filtering can be used in controllers for computer systems [11, 24, 26, 41].

Abstractly, Kalman filtering can be viewed as an algorithm for combining imprecise estimates of some unknown value to obtain a more precise estimate of that value. For example, usually both the prediction provided by the existing model and the new measurement are not precise, how should we extract useful information from these two values and obtain a more precise

one? We use informal methods similar to Kalman filtering in everyday life. For example, if one weight scale shows my weight is 160lb but another shows 155lb, I may estimate my weight to be 157.5lb and blame the weight scale for it's not being precise. We explored using Kalman filters for online updates of the ML models. Although many presentations of Kalman filtering exist in the literature, they are usually focused on particular applications like robot motion or state estimation in linear systems with Gaussian noise. This prevents users to understand the intuition of applying Kalman filtering to other problems. The key contribution of our work is to present the abstract statistical ideas behind Kalman filtering independently of particular application, and then show how these ideas can be applied to solve particular problems such as state estimation and model update. Our detailed derivation of Kalman Filtering from a much intuitive perspective can be found in [64].

3.3.3.5 Updating Machine Learning Models Online

For the Proteus project, we have implemented support for online machine learning using two different regression models. We have added support for regression using Multi-Layer Perceptrons and Random Forests. In the following, we briefly describe each model.

Multi-Layer Perceptrons

A multilayer perceptron (MLP) is a type of feedforward artificial neural network (ANN) that is primarily used for regression. MLPs are an example of a supervised learning technique, and use backpropagation for training. Like an ANN, a MLP is composed of one or more nodes in the input layer, one or more hidden layers, and an output layer. The backpropagation algorithm effectively learns the weights on the connections between the different nodes across layers. MLPs can model non-linear relationships between the dependent and the independent data variables.

Effectiveness for online updates. The backpropagation algorithm in MLPs make them amenable for online updates. The backpropagation algorithm is run for each batch of labeled data (backpropagation utilizes supervised learning), and updates the bias and weight values on each connection in the ANN. Many libraries and language packages like tensorflow, keras, and theano provide abstractions for easy development with MLPs. Most of these packages provide support to save the intermediate models to disk, and reload the models rapidly. Furthermore, these packages provide efficient implementations of ML models like MLPs by modeling ANN computations as sparse matrix-vector multiplications and making use of GPGPUs and advanced instruction set architectures available on recent computers. However, effective training of MLPs requires non-trivial training data which may not be available in all scenarios, especially during unit testing. Furthermore, the structure of the MLPs are very application-dependent and may require extensive tuning.

Random Forests Random forests are an ensemble supervised learning technique for regression (or classification). A random forest is an estimator that fits a number of regression (or classifying decision) trees on different samples from the data set, and averages the estimates from different trees to predict the final value. There can be a number of different policies on the basic technique of using Random Forests for regression.

Effectiveness for online updates. Random Forests improve over decision trees by avoiding overfitting to the training set. Random Forests are efficient algorithms and still provide good prediction capabilities even for non-linear relationships. Unlike MLPs, Random Forests can be reasonably trained with small data sets.

3.3.4 Task 3.5: Implementation of Relevant FAST Language Features and Testing Interaction with the Machine-Learning System

The FAST language, which was designed in Phase I, was realized in Phase II as an extension of Swift [5] focused on supporting adaptive streaming applications, and extended in Phase III to support adaptation as a first-class construct, as described in 3.3.7. FAST extends Swift with an instrumentation library and syntax for expressing application intent. The intent facility enables the user specify what it means for the application to behave well, and how the system should adapt to achieve this behavior. The intent is expressed in terms of designated program variables called *measures*, which hold values measuring application performance or quality as observed by the runtime. The instrumentation allows the user to specify variables called *knobs* in the code that can be tuned (modified) by the runtime to meet the specified intent.

Swift was chosen as the basis for FAST because it was specifically designed to support programming embedded applications, and strikes a good balance between expressiveness and performance. In particular, Swift

- supports modern programming paradigms such as object-oriented and functional programming;
- incorporates static type-checking and automatic storage management to detect and avoid many common bugs that plague embedded applications; and
- achieves predictable high performance due to its kinship with C (Swift is the direct descendant of Objective C) and its transparent foreign-function interface.

FAST extends Swift in three ways:

- a library with an instrumentation API;
- an intent specification language; and
- a runtime that periodically observes the measures and accordingly tunes the knobs so that the intent is met as the application processes its input stream.

The language and its runtime are documented in [65] , and the source code is available at the MIT GitHub [1].

3.3.4.1 Integration with Machine-Learning System

The control system in the FAST runtime relies on the models generated by the machine-learning system as it dynamically decides how to reconfigure the system. The machine-learning system generates these models based on the configuration space described in the knobs section of an intent specification. For each possible configuration described by the intent, a model contains an entry, which associates expected average measure values for that configuration.

One way to generate these models is through exhaustive *profiling*, where the application is executed for every possible configuration, and the observed measure values are recorded in the model. This approach becomes prohibitively expensive when the number of possible configurations is large. Since FAST supports controlling an application's state using *application knobs* corresponding to program variables, the number of possible knob configurations can be very large, making exhaustive profiling intractable.

To address this issue, the machine learning system takes an intent specification and a corresponding partial model consisting of profiles for a subset of the configurations described in the intent specifications, and extrapolates entries for the remaining configurations using machine-learning to construct a regression model approximating the map of knob configurations to measure values. This approach is used offline, before execution, to extend a small profile-based model to a comprehensive model covering all possible configurations. As described in Section 3.3.5, this approach also serves as the foundation of the Proteus online machine-learning facility.

3.3.5 Task 3.6: Offline and Online Machine-Learning Algorithms, Model Refinement, and Intent-Aware Programming for Demo Preparation

After Phase II and during Phase III, FAST was gradually integrated with the Machine Learning (ML) module using a ReSTful interface. The interface is a means to enable the ML module to update its *potentially stale* regression models, and forward the updated predictions (i.e., measures) of different knob configurations to the Proteus controller model.

The interface, supported by both the FAST runtime and the machine learning component, consists of two end-points: `/setup` and `/update`, described in Table 4.

The integration adheres to the following protocol:

1. At the beginning of execution, before any inputs have been processed, the FAST application calls the `/setup` end-point of the machine learning component, with a description of the total set of configurations, and an initial controller model (comprised of a knob table and a measure table) that maps a subset of the total set of configurations to their expected measure values. The model type is specified in a *.json* file. For example, there are six available types of models that can be built, including performance, quality, energy, energy_delta, bitrate, latency and power. Only models that are desired by the FAST runtime are built. In addition, there are several supported online machine learning methods to build models, including random forest, Gaussian process, multi-layer perceptions and

Table 4: ReST Interface of the Online Machine Learning Module

End-point	Description
/setup	Request: The FAST runtime initiates the request <i>once</i> at system start. 1. A dictionary from knob name to knob range and reference value. 2. An initial knob table, that is, a table that lists the knob values of a set of configurations. 3. An initial measure table, that is, a table that lists the measure values of the set of configurations listed in the initial knob table. For example, this data can be from the offline profile phase. Response: The ML module responds to the request with an initial model computed based on the training set. 1. A knob table with entries for <i>each</i> of the configurations that are possible given the knob dictionary passed in the request. 2. A measure table with entries for all configurations listed in the response knob table, sampled from the offline-trained machine learning model corresponding to the application.
/update	Request: The FAST runtime initiates this request periodically during application execution. 1. The current knob table, that is, a table that lists the knob values of all possible configurations specified by the current intent. 2. A measure table that includes the measure values of each of the configurations that the system has entered, either since the beginning of execution (if this is the first call to /update), or since the last call to /update, that will be used to update the machine learning model. Response: An updated model based on both the training set and feedback, collected at runtime and sent in the request. 1. A knob table with entries for each of the configurations that are possible given the knob dictionary passed in the request. 2. A measure table with entries for all configurations listed in the response knob table, sampled from the offline-trained machine learning model corresponding to the application.

linear regressions. Different models provides different accuracy and speed, which serve as a trade-off for various workloads. The `/setup` end-point returns a controller model with the missing configurations filled-in by estimates sampled from the offline machine-learned model.

2. During execution, when the anomaly (which will be introduced later) detector flags that the model needs to be updated, the `/update` end-point is called with the current knob table (the part of the controller model that lists the knob values of all the possible configurations), as well as a table that contains the measure values of all the configurations that have been observed since the last time the controller model was updated. These values are used to update the ML models online, and then `/update` returns a new measure table based on this updated machine learning model. There are several ways to merge the new measurement table with the old table. In our ReST interface, overwrite, summation and weighted average is supported for various application scenarios.

The anomaly detector is implemented as part of the controller component, and flags that an update is needed when oscillation in the controlled measure is detected. The amount of oscillation allowed depends on the application and its the hardware. For example, in x264:

- On the same hardware and with the same application knob settings, a higher-resolution version of the same video sequence yields lower performance than a lower-resolution version
- Using the same video with the same application knob settings, more powerful hardware yields better performance.

In our implementation, the anomaly detector summarizes the amount of oscillation every some number of frames (40), in order not to report noisy measurements too frequently. According to the current video configuration and hardware setting, it is based on a pre-set and application-dependent threshold. If the oscillation exceeds the threshold, the new measurement will be sent to the online machine module for model refinement. As described above, the `/update` end-point will send back the refined model to the FAST runtime for future control.

3.3.6 Effectiveness of Online Machine Learning Algorithms

During Phase III, the online machine learning module has been successfully integrated with the FAST runtime despite much technical difficulty. The FAST runtime is able to detect the oscillation and send new measures to the online machine learning module while the online machine learning module is able to refine the model adaptively and send it back to the FAST runtime for future control.

The effectiveness of the online machine learning module is showed in Figure 22. The target workload used in this example is x264. Note that x264 is one of the most challenging applications to control dynamically because its behavior is heavily dependent both on configuration and on inputs. In the FAST runtime, the controller performs control based on an offline profiling on representative inputs. The main limitation of the controller is that the profiling only takes knobs into consideration without taking into account input features. The controller can start to misbehave if the current input is not well represented by the measure table during inference.

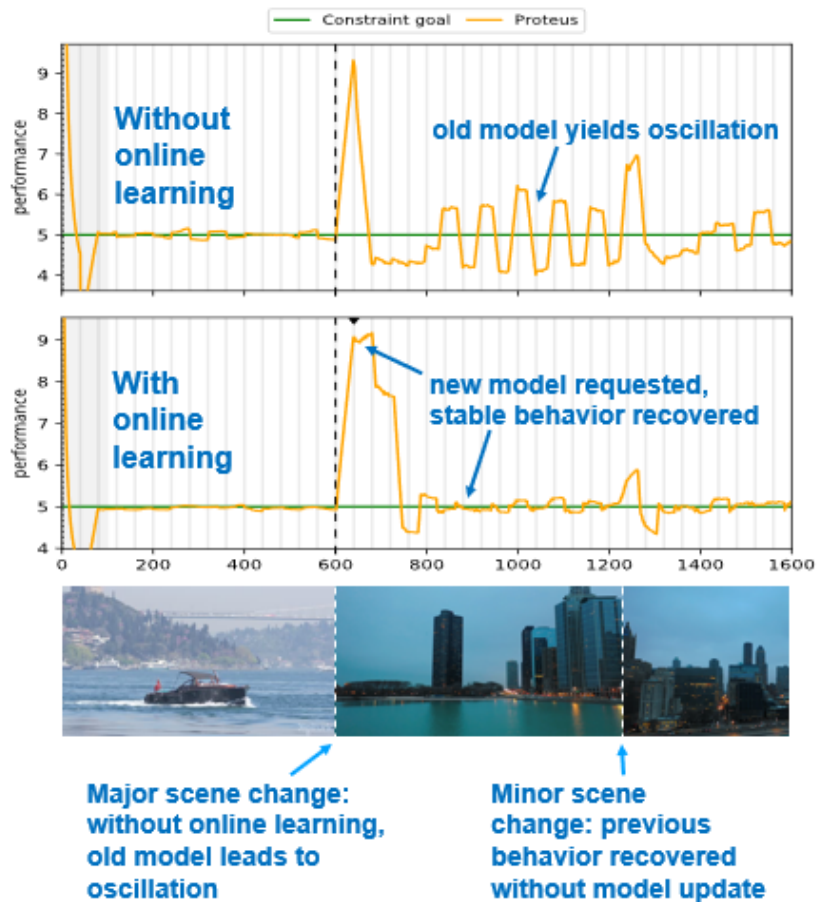
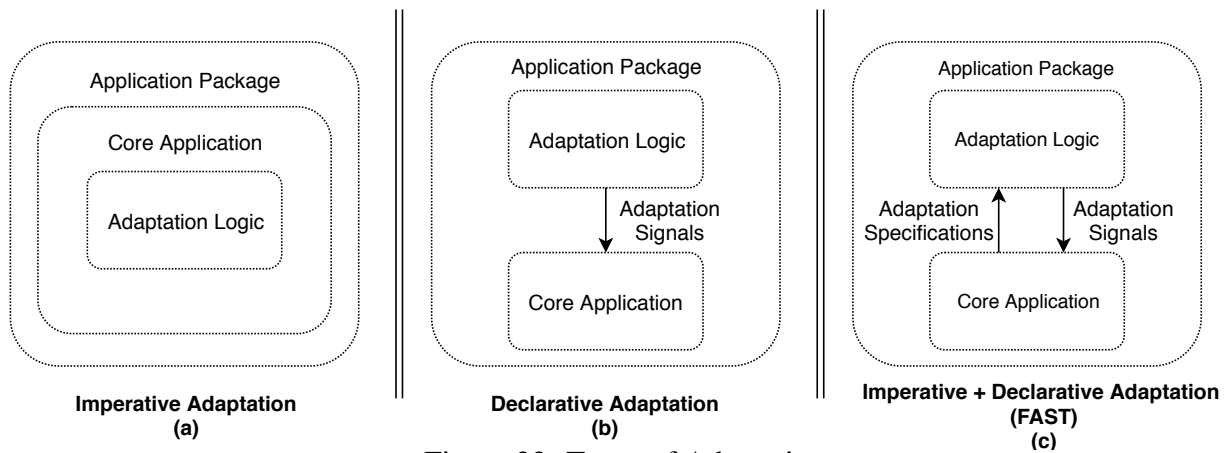


Figure 22: Effectiveness of the Online Machine Learning Module

In such applications, good behavior can be recovered by online learning, where a new model, learned from recent data, replaces a model that is no longer representative of the current input stream.

In this experiment, the scene is constructed by concatenating two videos of different scenes. The first video records a boat sailing in the ocean while the second one contains views of a modern city. Though the two videos are of the same resolution, the performance numbers for these two videos are very different. The offline profiling is conducted on the first video. In Figure 22, the first 600 frames correspond to the first video and the later 1000 correspond to the other video. We set the goal to 5. In the first figure in Figure 22, we can see that the controller alone is able to meet the goal for the first video. However, as soon as the video hits the major scene change, the controller oscillates considerably when the encoder begins processing the second video. Since the model cannot be updated in this case, the controller can do nothing about it. In contrast, in the second figure, the online machine learning kicks in after the FAST runtime detected large oscillations. The online machine module sent back the updated model at frame 680 and we can see that the oscillation will no longer occur in the second phase of the video. The x264 application is still able to meet the intent set by the user in dynamic environments. In principle, this technique is also applicable to other applications whose performance may vary a lot between inputs.



In terms of model construction time, it depends on the method used and the size of the model. Typical construct time for random forest models on 100 knob entries takes around 0.5 seconds, which can be considered reasonable for applications based on x264.

3.3.7 Adaptation as a First-class Programming Construct

Logical correctness is no longer the only guarantee that modern software systems are expected to provide. The Proteus project recognizes that mission critical systems must also deliver predictable performance, accuracy, and energy. Further, while those are the dimensions directly handled in this project, we are aware that many other constraints can affect real systems including: bandwidth, memory consumption, security, temperature, etc. While developing software to meet such diverse requirements is already a challenge, the problem is further complicated by the fact that all of these quantifiable metrics are affected not just by the application code, but also by factors beyond program control like the operating environment and input workload. Successful software deployment now requires applications that are both logically correct and capable of *adapting* to deliver predictable behavior even in unpredictable dynamic environments. Indeed, the overarching goal of the Proteus project has been to support that adaptive capability in a principled way by creating a rigorous engineering approach to construction of adaptive computer systems.

In prior work, adaptive software takes one of two forms: *imperative* or *declarative*. In imperative approaches (Figure 23a), the *adaptation logic* is inter-mixed with application code that provides core functional behavior. Furthermore, imperative approaches rely on programmer-defined rules and heuristics to guide adaptation. In declarative approaches (Figure 23b), application developers declare a program's *goals* and *knobs* whose configuration dynamically alters quantifiable program behavior, but some independent adaptation module decides how to set the knobs to meet the goals at run time. Declarative approaches provide a separate adaptation logic that operates (usually in a runtime) outside of the core program functionality, allowing developers to focus on their application domain and delegate adaptation.

Imperative adaptation—based on heuristics and rules—has the benefit of putting the application in full control of all adaptation; *i.e.*, it does not rely on some independently developed module to produce the correct quantifiable behavior. Designing robust heuristics and rules to handle all goals and situations that software might encounter is quite difficult, however. Furthermore,

imperative adaptation makes software unwieldy and harder to maintain as embedding adaptation logic in the core program violates separation of concerns by mixing core functionality with code for maintaining quantifiable behavior.

Declarative adaptation introduces a different set of benefits and drawbacks. While such approaches are quite robust and achieve good separation of concerns, they do not allow the core program to communicate with and/or directly manipulate the adaptation logic after it is initialized. Thus, if the program's desired behavior or adaptation constraints change at runtime there is no mechanism for communicating these changes to the adaptation logic. Using existing declarative approaches, the program would have to destroy the adaptation module and reinitialize it to meet different goals or to tune a different set of parameters. During re-initialization the application will likely fail to meet its goals.

Both imperative and declarative approaches have significant benefits and intuitively they should be combined. As shown in part (c) of 23, the application should have complete control over adaptation *even after the adaptation module starts* (as in the imperative approach), but the fine-grain management of tunable parameters should be delegated to separate, dedicated logic that can be independently validated, debugged, and reused (as in the declarative approach). To realize the benefits of both, Proteus has *implemented adaptation as a core feature of the programming framework*.

Achieving this goal has demanded significant advances over existing declarative adaptation frameworks. While any adaptation framework will provide constructs for measuring behavior and tuning configurations, that support alone is not enough. We argue that an application's *adaptation policies*—or *intent*—should be specified in a domain-specific language as a constrained optimization on expressions involving the measures and knobs. Behavior that cannot be violated (e.g., latency requirements) should be expressed as a *constraint* while behavior that is to be minimized or maximized (e.g., accuracy) is an *objective*. The application should be able to dynamically alter goals, allowable knob configurations, and even the adaptation policy itself while the application (and adaptation logic) is running. To the best of our knowledge, *no other system provides such a general framework for specifying and dynamically altering adaptation as first-class programming constructs*.

In the Proteus project we have implemented the aforementioned ideas as a proof-of-concept—FAST—that extends Swift. FAST programmers use a novel domain specific language to define an *adaptation specification* that relates the constraints and objectives to programmatically defined knobs and measures. The FAST runtime uses the adaptation specification to compute a *schedule* of knob configurations consistent with the specified adaptation policy. In contrast to prior work, FAST's adaptation specifications are *general*; *i.e.*, they may comprise arbitrary arithmetic expressions over the measures and knobs. In addition, FAST's adaptation specifications are *dynamic*; *i.e.*, the program can dynamically update them at runtime. Both the generality and dynamism of FAST's adaptation specification system distinguish it from prior work.

To demonstrate the benefits of adaptation as a first class programming construct, we reimplement six adaptive applications from the literature—including both embedded and server examples—using FAST. We emphasize that these represent additional case studies beyond the challenge problems required for the BRASS project. These case studies illustrate a lack of flexibility provided by existing declarative approaches and show how FAST provides those using natural interfaces. These case studies also show how FAST provides a uniform programming interface for developing applications that run across different kinds of systems operating under

different conditions and requirements.

More detail on the FAST language extensions, the FAST adaptation specification language, and case studies demonstrating the approach can be found in [65].

4 Results and Discussion

This part discusses tasks related to the evaluation of the Proteus system and target platform, which cut across the task areas described in Part 3. This includes the definition and evaluation of challenge problems (Section 4.1.1), testing of challenge problems on the Weak Memory Model architecture (Section 4.1.2), definition of metrics for evaluation (Section 4.1.3), porting of the Proteus system to different platforms (Section 4.2.1), and the applications that were used to demonstrate Proteus throughout the program (Section 4.2.2).

4.1 Evaluation

4.1.1 Tasks 4.1, 4.2, 4.3, 4.4, 4.5 and 4.9: Challenge Problems for Phases I, II and III

The Proteus challenge problems comprised four different applications:

1. **SAR** A *Synthetic Aperture Radar* (SAR) signal processing simulation.
2. **x264** The *x264* video encoder.
3. **CaPSuLe** A streaming image matching application based on an approximate matching engine system, which uses locality-sensitive hashing to conserve energy.
4. **flightTestScenario7** An adaptive flight test radio frequency scheduler capable of dynamically reallocating bandwidth subject to constraints, while optimizing a dynamic system value metric.

4.1.1.1 SAR: Synthetic Aperture Radar

A radar system detects objects using reflected electromagnetic waves and reports their position or speed. Multiple measurements from a radar system can be used to create an image of the objects in a given angular region. The ability of the radar to distinguish two nearby features in an object or surface is commonly referred to as *distance resolution*. We separate *range (depth)* ρ_r , and *azimuth (width / beam)* ρ_a resolutions. The azimuth resolution of the image depends on the aperture (length) of the receiving antenna of the radar system. However, in some scenarios, it is technically difficult (and expensive) to dynamically change the size of the radar antenna. Fortunately, it is possible to simulate a physically large antenna using a much smaller antenna by mounting it on a moving platform and using signal processing to synthesize the image that would be generated by an antenna of larger aperture. The obtained *synthetic* aperture depends on the speed of the platform and the time during which the target reflects waves back to the radar antenna; the framework is referred to as *Synthetic Aperture Radar* [16]. Details of this challenge problem are given in Appendix 5.2.3.

4.1.1.2 x264: Video encoding

x264 is a video codec in accordance with the H264 standards.

The standard *H.264: Advanced Video Coding* was published by the international standards bodies ITU-T and ISO/IEC. [44]. It describes a format to be followed when encoding/decoding video sequences, the specifics of encoding are not given, the standard states it should be in accordance with the decoding steps. *H.264* provides better image quality and compressed bit rate than *MPEG-2* or *MPEG-4*, though, at a higher computational cost.

The main stages of *H.264* are:

Predict: Inter– or Intra–Prediction is applied on the frames. Then, residuals are obtained from comparing the prediction to the actual frame.

Transform: Block–based or Image–based transformations are applied to the residuals. The stage concludes by Quantization.

Encode: The data is encoded by variable length coding and/or arithmetic coding.



Figure 24: H.264 Pipeline

Details of this challenge problem are given in Appendix 5.2.3.

4.1.1.3 CaPSuLe: Image matching

CaPSuLe, the application that CP3 is based on, is a new and highly energy and computation efficient image matching algorithm developed by Rice University and Seoul National University researchers [37]. The algorithm is implemented in C++. For Proteus, CaPSuLe is implemented as a Swift wrapper over the existing C++ code [66]. The test parameters of CP3 are fully documented in Appendix 5.2.3. The complete Swift source code of CP3 is available at the MIT GitHub [2].

4.1.1.4 flightTestScenario7: Radio Bandwidth Scheduling

This challenge problem was based on a test scenario provided by SwRI, where three “test articles” (airplanes) share radio communications infrastructure on a test range. Through a custom application, based on SwRI’s OrientDB database and interface, Proteus observes measures such as the communications queue depth of each test article, as well as a system value measure that SwRI is developing to estimate the value of the currently ongoing tests.

Proteus dynamically controls the allocation of bandwidth to each test article, as well as its corresponding ground station. During the test, one or more of the test articles may suddenly stop using its allocated bandwidth, for example because it flies behind a mountain range. This makes additional bandwidth available to the remaining test articles. Because Proteus intents are optimization problems, this makes it possible to solve the bandwidth scheduling problem in

a way that optimizes the system value, which is something the current scheduling algorithms used by SwRI are not capable of.

Perturbations, that is, when a test article stops actively using its allocated bandwidth, correspond to uses of the new Knob APIs, introduced in FAST in Phase III, that restrict its bandwidth value to zero, and then restore full control of the knob when the test article again becomes available.

This challenge problem also makes use of the ability to impose static constraints on knobs that was also introduced in FAST in Phase III, which eliminate configurations where the total bandwidth used by the test articles exceeds the total available bandwidth.

The system value that Proteus should maximize is the objective function of the intent. Whenever the value associated with a test article's radio communications changes, the intent is updated to reflect this.

Details of this challenge problem are given in Appendix 5.2.3.

4.1.2 Task 4.6: Testing of Proteus with Integrated Memory Models on Hardware Drone

The Weak Memory Model developed under the Proteus project is documented in Section 3.2.3. This section describes the empirical evaluation of the memory model, and lists conclusions from this research.

Empirical Evaluation of Performance

Based on the theoretical analysis alone, it is difficult to predict what the practical impact of a switch to WMM will be on processor performance. Although WMM may require additional fences in the software, they do not necessarily lead to worse performance during execution. In fact, it is possible that switching to WMM could improve performance by allowing more flexibility in hardware optimizations and simpler hardware.

First, we evaluate the performance impact caused by the same-address load-load ordering and the load-to-store ordering in WMM. The impact of same-address load-load ordering and load-store ordering will be on *single-thread* performance. Therefore, we study the performance of a single processor of the following four memory models using the SPEC CPU2006 benchmarks. We modeled and out-of-order (OOO) processors in GEM5 [12]. Table 5 shows the detailed parameters; the sizes of the important buffers (ROB, load buffer, store buffer and reservation station) are chosen to match a Haswell processor.

We run all reference inputs of all SPEC CPU benchmarks (55 inputs in total) in full-system mode. For each input, we simulate from 10 uniformly distributed checkpoints. For each checkpoint, we first warm up the memory system for 25M instructions, then warm up the processor pipeline for 200K instructions, and finally simulate 100M instructions in detail. We summarize the statistics of the 10 checkpoints to produce the final performance numbers for this benchmark input.

Load-to-Store Ordering: The load-to-store ordering in WMM will prevent a store from being sent to memory until all older loads have been executed. Theoretically, by removing this constraint, a store can be retired from the store queue earlier, and performance may benefit by reduced stalls due to a full store queue. Figure 25 shows relative performance improvement

Table 5: Processor Parameters

Single core @2.5GHz with x86 ISA (modified O3 CPU model)	
Width	4-way fetch/decode/rename/commit, 6-way issue to execution, 6-way write-back to register file
Function units	4 Int ALUs, 1 Int multiply, 1 Int Divide, 2 FP ALUs, 1 FP multiply, 1 FP divide and sqrt, 2 load/store units
Buffers	192-entry ROB, 60-entry reservation station, 72-entry load buffer, 42-entry store buffer (holding both speculative and committed stores)
Classic memory system with 64B cache lines	
L1 inst	32KB, 8-way, 4-cycle hit latency, 4 MSHRs
L1 data	32KB, 8-way, 4-cycle hit latency, 8 MSHRs
Unified L2	256KB, 8-way, 12-cycle hit latency, 20 MSHRs
L3	1MB, 16-way, 35-cycle hit latency, 30 MSHRs
Memory	80ns (200-cycle) latency and 12.8GB/s bandwidth

by *not* enforcing load-to-store ordering for each benchmark and for different store queue sizes (SQ42 for 42-entry store queue, SQ28 for 28 entries, SQ14 for 14 entries, and SQ7 for 7 entries). The last column in the figure is the average across all benchmark inputs. As we can see, when the store queue size decreases, the benefits of removing load-to-store ordering will increase. However, the performance improvements are still negligible (about 0.5% on average for a 7-entry store queue) and never exceed 2.5%.

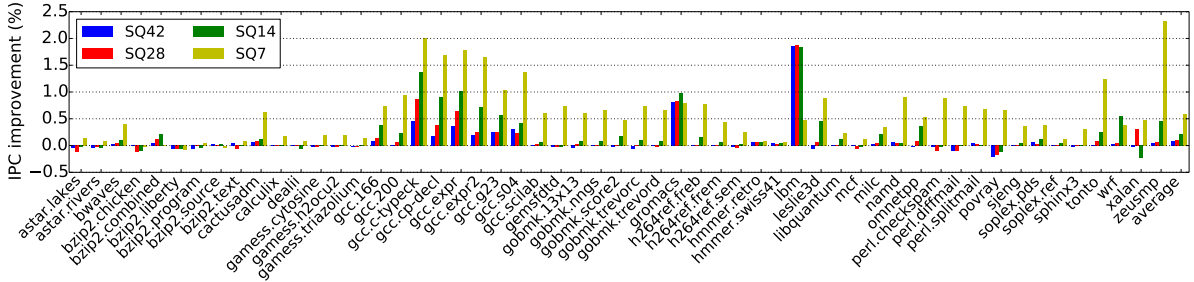


Figure 25: Performance Improvement by Removing Load-to-Store Ordering

Same-Address Load-Load Ordering: The same-address load-load ordering constraint in WMM will impact performance in the following two ways:

1. **Kills:** when a load L computes its address, the processor kills any younger load which has finished execution but has not got its value from a store younger than L .
2. **Stalls:** when a load L is ready to issue to start execution, if there is an older unissued load for the same address and L cannot get forwarding from any store younger than the unissued load, then L will be stalled.

If we do not enforce such ordering, neither the above kills nor the above stalls will happen. Figure 26 shows relative performance improvement by *not* enforcing same-address load-load ordering for each benchmark. The last column in the figure is the average across all benchmark inputs. As we can see, the performance improvements are all negligible (less than 0.3% on average) and never exceed 2.5%.

by these applications. Radar uses fine-grained parallelism and synchronization through shared-memory data structures. x264, on the other hand, uses course-grained parallelism with very little synchronization at all.

These results contradict the traditional wisdom that using weak memory models hurts performance. In fact, the performance impact depends heavily on the exact application characteristics and hardware implementation. In this case, we have carefully optimized both the WMM and TSO implementations and the additional hardware overhead of implementing TSO outweighs the extra software overhead required for WMM. Since this is application dependent, much more study would be required to draw general conclusions. However, this result shows that using weak memory models like WMM is not necessarily bad for performance and can, in fact, improve overall performance as well as software longevity and hardware simplicity.

Resilience to Future Hardware Changes

WMM allows all kinds of re-orderings except a store overtaking a load (i.e., load-store reordering). Forbidding load-store reordering is in line with the fact that most processors commit instruction in order to implement precise exceptions and interrupts. The variety of re-orderings allowed by WMM gives abundant flexibility in hardware implementations. Furthermore, WMM does not enforce dependency ordering, and thus allows hardware to perform all kinds of speculations, such as branch prediction, memory dependency prediction, and even load-value prediction. This feature enables WMM to encompass future microarchitectures with deeper speculations. We have proven that WMM can be implemented using cache-coherent memory systems and out-of-order processors with all three speculations mentioned above. This proof also provides insights to architects when they try to evolve the microarchitecture.

4.1.3 Task 4.7: Integrated System Validation Through Metrics

The FAST intent specifications that designers use to express their intent to Proteus encode *constrained optimization problems* that consist of two parts: a *constraint* over one or more measures, and an *objective* that consists of an objective function and an optimization type. To satisfy the specification means to (approximately) satisfy the constraints, while optimizing the objective function with respect to the optimization type. Solutions that provide a better objective function, but do not satisfy the constraint, cannot be said to meet the designer's intent. This priority of the constraint over the objective should be reflected when quantifying the plasticity of Proteus compared to a baseline system.

4.1.3.1 Mean Combined Relative Constraint Error (MCRCE)

The combined mean relative error is a summary metric that represents the degree to which the system is able to meet the constraints encoded in the active intent specification, accumulated across all iterations of a test execution.

At each iteration, the relative error is computed for each constraint. Proteus supports two kinds of constraints: a single equation; a conjunction of one or more inequalities. For the single-constraint case, the error is the absolute relative error versus the constraint goal. For

the multi-constraint case, the error in each dimension is 0 when the inequality is satisfied, and otherwise the absolute relative error versus the constraint goal. The *combined* error for the current iteration is the average of the errors in each dimension. The MCRCE is defined as the average combined error across all iterations of the test execution.

Note that this definition remains meaningful also when the intent specification, and thus the constraints, change during the course of the test. This is crucial to support evaluation when tests include perturbations that dynamically modify intents. The MCRCE is used in two main contexts: it is compared to a constraint error threshold while computing the verdict for a particular test case (details can be found in Appendix 5.2.3), and it is used to compare the constraint granularity gain plasticity metric described in the following section.

4.1.3.2 Plasticity

The plasticity of a system is a measure of its ability to adapt to changes. Existing systems can be adapted to the conditions they are expected to operate in during execution, by setting their configuration parameters before the mission begins. To ensure that the system can meet specific constraints, such as on performance, the system designer may configure the system in a way that ensures that these constraints are met despite changes in operating conditions, such as the ambient load on the system. When a system is not capable of dynamic adaptation, enforcing such constraints comes at the cost of other possible objectives, such as keeping energy consumption low. A correct definition of a baseline must therefore be derived from the designer's intent, that is, the FAST intent specification.

We define two metrics for plasticity: 1) constraint granularity and 2) unified knob granularity, each with its own baseline against which the gain is computed.

Constraint Granularity Gain The baseline for computing the constraint granularity gain is the set of fixed configurations that are close enough to satisfying the constraint in their corresponding runs. Let B be such a set. Let $M(i)$, the achievable measure space size at iteration i , be the number of ways that the constraint can be satisfied by Proteus using the available configurations. Further let N be the number of iterations in a Proteus run. We define the constraint granularity gain C of Proteus for a given execution with N iterations as:

$$C = \frac{\sum_{i=1}^N M(i)/|B|}{N} \quad (8)$$

These quantities $|B|$ and $M(i)$ are computed as follows.

Baseline runs Non-adaptive systems can only use one particular fixed configuration during a run. The MCRCEs defined in Sections 4.1.3.1 measures how well a run is meeting the given constraint. The smaller the MCRCE, the better. Let mce be the minimum of all the MCRCEs of all the fixed configuration runs. A fixed configuration run whose MCRCE is equal to mce is one that best satisfy the constraint. However, such a fixed configuration run may not yield an optimal objective. There may exist other fixed configuration runs with greater MCRCEs that better optimize the objective. We can obtain a larger set of candidate runs by relaxing the error bound for meeting the constraint. Instead of mce , we consider constraint error bounded by $bf * mce$, where bf is some constant factor greater than one. We choose $bf = 2$, as anything larger

Approved for Public Release; Distribution Unlimited.

than 2 would allow too large a constraint error. The baseline runs for constraint granularity is the set of fixed configuration runs whose MCRCEs are less than $2 * mce$.

Proteus achievable measure space At each iteration in a run, Proteus examines pairs of fixed configurations from the cross product, $A \times B$, of two disjoint sets of fixed configurations and selects an appropriate pair to build a schedule for switching between configurations. The schedule is such that the constrained optimization problem is solved on average over a window of execution. The Proteus achievable space size is thus $size(A) \times size(B)$. The two components, A and B , of this cross product for the single constraint case differ from those of the multi-constraint case. They are defined as follows.

Let C be the set of configurations in the active model.

- single constraint:

Let g be the constraint goal, let $m(c)$ be the constraint measure value for a configuration c , define $A = \{c \in C | m(c) > g\}$, and $B = \{c \in C | m(c) \leq g\}$.

- multi-constraint:

Denote $p(\vec{m}(c), \vec{g})$ as the multi constraint predicate on a vector of measure values in a configuration c with constraint goal $\vec{g}$, define $A = \{c \in C | p(\vec{m}(c), \vec{g})\}$, and $B = \{c \in C | \neg mcp(\vec{m}(c), \vec{g})\}$.

Unified Knob Space Granularity Gain Plasticity metrics quantify the degree of adaptability of a platform/application. One aspect of plasticity is the size of the configuration space. Compared to the baseline of existing systems, Proteus increases multiplicatively the size of the space of configurations available for dynamic adaptation and, as a result, offers multi-fold plasticity with gains in application fidelity. This increase in the size of the configuration space corresponds to an increase in the size (volume and/or granularity) of the space of achievable goals, as defined by the programmer's intent. Let AC be the set of application configurations. Let SC be the set of system configurations.

Existing frameworks for dynamic adaptation can exclusively tune either application configuration or system configuration, but not both. Thus the size of their configuration spaces is $size(AC) + size(SC)$

In contrast, Proteus is capable of controlling both application configuration and system configuration dynamically, enabling adaptation to changing resources and operating environment at runtime. Thus the size of Proteus configuration space is $size(AC) \times size(SC)$.

The unified knob space granularity gain is defined as

$$\frac{size(AC) \times size(SC)}{size(AC) + size(SC)}$$

4.2 Porting and Demonstration

4.2.1 Task 4.8: Porting and Platform Validation

4.2.1.1 Porting

Table 7 shows examples of platforms to which the Proteus system was ported over the course of the project, along with one or more of the challenge problem applications described in Section 4.1.1. The diversity of hardware and operating system environments is a testament to the portability of the software implementation and the robustness of the approach to adaptive computing that is embodied in Proteus.

Table 7: Examples of Platforms to Which Proteus was Ported

Platform	Architecture	Operating System
Xilinx ZCU102	ARM Cortex-A53	Ubuntu Linux 16.04
Odroid XU3	ARM Cortex-A15	Ubuntu Linux 16.04
AWS c5.xlarge	x86	Ubuntu Linux 16.04
Dell PowerEdge T430	x86	Ubuntu Linux 16.04
ThinkPad T560	x86	Ubuntu Linux 19.10
MacBook Pro 2017,2018	x86	macOS 10.13-10.15

4.2.1.2 Platform Validation

Extensive internal tests were conducted for each challenge problem, and the test results were validated by collecting data using the SwRI test infrastructure. For Phase III CP1, this involved integration with the SwRI RadioQueue simulator (see Figure 30). For Phase III CP2 and CP3, the Proteus emulator for the Xilinx ZCU102 was used to virtualize the hardware platform for the machine-learning drone. To formalize the evaluation, SwRI were provided with a test plan (5.2.3) for each of the CPs, as well as a Python implementation of the test case generators and evaluation scripts. Figure 27 shows an example graphical overview of tests executed on the SwRI test infrastructure.

For each challenge problem, a pie chart breaks down the tests into the different test cases, as well as into cases whose outcomes were expected and those whose outcome was unexpected, for example because the aggregate error (4.1.3.1) exceeded the threshold set for that challenge problem. Histograms of aggregate error, objective function values, and plasticity metrics are also provided, as well as a list of causes for all tests with unexpected outcomes.

There are some tests with unexpected test outcomes for CP2 and CP3. As we have discussed previously, these results are due to the choice of “Constraint error threshold” and “Objective function error threshold” for each respective challenge problem, as illustrated by the vertical red line in the histograms at the bottom of the attached evaluation report screenshot. For example, the threshold for CP2 was chosen so that the vast majority of tests produce an error below the threshold. Choosing higher threshold could produce a 100% pass rate, but would misrepresent the accuracy that the adaptive application achieves in the different scenarios. Choosing a

lower threshold would better represent that accuracy (as the distribution of “Proteus cumulative total relative constraint error” shows), but would yield a larger number of spurious unexpected test outcomes, necessitating manual validation of a larger number of tests.

4.2.2 Task 4.10: Demonstration

4.2.2.1 Phase I Platform Workshop Demo

For the Phase I Platform Workshop, we created a demonstration based on Phase I Challenge Problem 2 (described in Section 5.2.3). The demo involved realtime H.264 encoding of video on the ZCU102 prototype hardware platform, as you might find on an autonomous vehicle. Raw un-encoded video (stored on the platform) was encoded using CP2 (x264) under Proteus control and streamed across the network to a laptop running a simulated base-station user interface (Figure 28). This interface displayed the output video and allowed the user to control and observed various parameters of the simulated system.

4.2.2.2 Phase II Platform Workshop Demo

For the Phase II Platform Workshop, we created a new demonstration based on the demo from Phase I. The demo executed on the same hardware platform, but exposed a new user interface (Figure 29). The most significant changes for Phase II were that we used the new FAST implementation of x264 and the user interface allowed the user to edit and send new FAST intent specifications.

Also new for Phase II was the ability of Proteus to control system knobs such as core frequency and number of cores utilized. In Phase I, these knobs were not under Proteus control but were instead manually controlled by the user to simulate changes in the environment. This change exposed a weakness in our platform’s implementation of the system knob interface, namely that it took too long to make changes to the knobs. Under Proteus control, the knobs are adjusted much more frequently and the time it takes to make a change effects the responsiveness of the system. Therefore, we heavily optimized this code, eliminating all redundant operations and pre-computing as much as possible. As a result, we were able to speed up system knob changes significantly and improve the smoothness of our demo when Proteus decided to exercise them.

4.2.2.3 Phase III Platform Workshop Demo

Flight Test For the Phase III Platform Workshop, we worked with SwRI to develop the first Phase III challenge problem, `flightTestScenario7` (5.2.3) into a live demonstration. The `flightTestScenario7` application executed in tandem with the SwRI radio queue simulator, communicating through the OrientDB graph database. The demonstration showed a scenario where bandwidth relinquished by an airplane was reallocated dynamically to the remaining airplanes, showing that system value can be recovered by changing the bandwidth allocation dynamically based on the values of the feasible schedules.

Figure 30 shows the graphical user interface of SwRI’s radio queue simulator (on the right) visualizing the bandwidth allocations selected by Proteus (top-left), communicating through



Figure 27: Proteus Phase III Evaluation Report

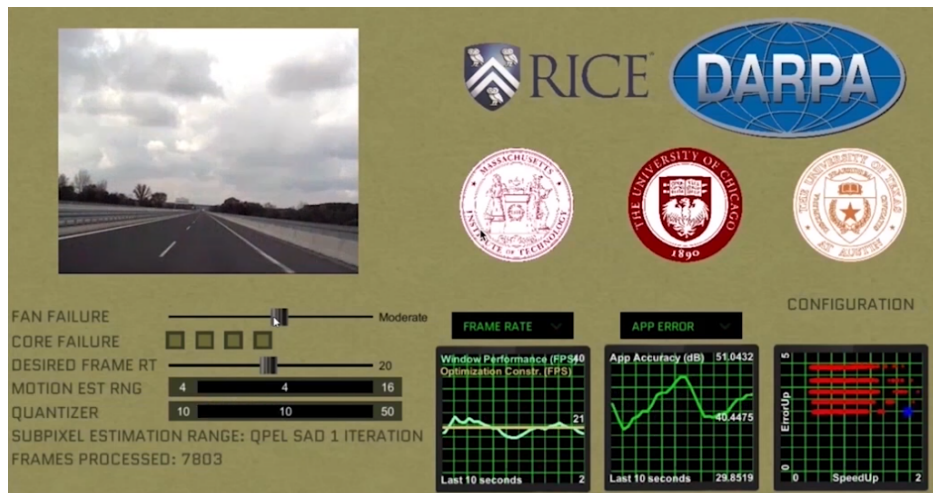


Figure 28: The Graphical User Interface of the Phase I Demonstrator Application

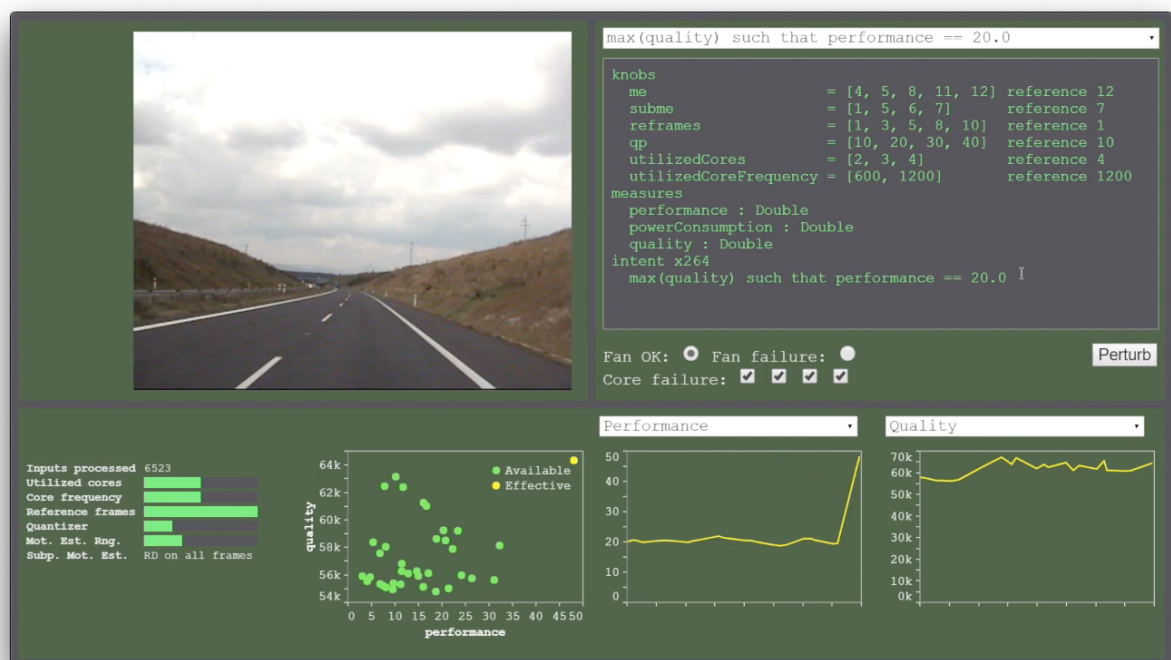


Figure 29: The Graphical User Interface of the Phase II Demonstrator Application

OrientDB (bottom-left).

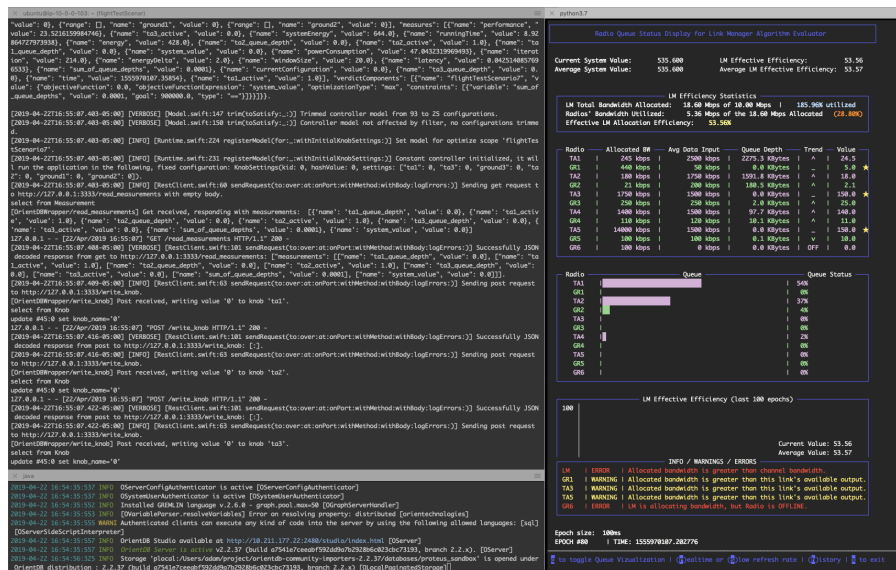


Figure 30: SwRI's Radio Queue Simulator Showing Bandwidths Selected by Proteus

Prediction of Chaotic Dynamical Systems The Phase III demonstration also included a live demo of prediction of chaotic dynamical systems using a neural network architecture called Echo State Networks [15].

5 Conclusions

5.1 Lessons Learned

5.1.1 Solving Constraint Optimization Problems

For our Capri work, we realize that although there is a large body of work on using approximate computing to reduce computation time as well as power and energy requirements, little is known about how to control approximate programs in a principled way. Previous work on approximate computing has focused either on showing the feasibility of approximate computing or on controlling streaming programs in which error estimates for one input can be used to reactively control error for subsequent inputs. We are able to address the problem of controlling tunable approximate programs, which have one or more knobs that can be changed to vary the fidelity of the output of the approximate computation. We learned how the proactive control problem for tunable programs can be formulated as an optimization problem, and then gave an algorithm for solving this control problem by using error and cost models generated using machine learning techniques. Our experimental results show that this approach performs well in controlling tunable approximate programs.

5.1.2 Handle Input-Dependence through Online Learning

Over the course of the project we used several challenge problems that exposed our system to different challenges. Out of these, the application we found to be most difficult to control was the x264 video encoder, whose behavior across all its measures varies a lot depending on various properties of the part of the stream that is currently being processed. Some kinds of scene changes can be accommodated by feedback control, or by state estimation, where for example an ambient load is making the configurations slower across the board. But if the relationships between the configurations change to the point where, for example, one configuration that used to be slower than another is now faster, then the model needs to change. We implemented this in a separate module that receives a sample of the recent measure values as input, fuses this with the data that was used to generate the initial model offline, and generates a new model that replaces the bad model. Figure 22 shows the behavior of processing a video sequence that consists of two concatenated scenes, the first of which was used to obtain the initial model. We can see that, without online learning, the controller is still able to stay fairly close to the goal, but the performance oscillates as the controller struggles to cope with the incorrect model. When we let the online learner replace the model a short while after the controller starts oscillating, stable behavior is regained after the scene change.

5.1.3 Model Size

Model size plays an interesting role in combining online learning and offline learning. Larger model size is not a problem in offline control (though it may take longer to build) and it can potentially improve the accuracy of the model. However, larger models increase the time for the online learning to react during execution. Since an application runs one knob setting at time, it takes long time for the online learning module to fully update the whole model, even when extrapolation is used in model construction. In this project, we chose a moderate model size in order to trade off accuracy and profiling/training time.

5.1.4 Modularity Through High-Level Interfaces

A design principle that brought with it a lot of benefits was the adoption of high-level interfaces wherever possible. On the side of the programmer interface, this took the form of the programming language and library FAST, that Krishna spoke about, where a small number of simple abstractions hide the complexity of monitoring, scheduling, and actuation that are needed to achieve the adaptive behavior of Proteus from the user. In the implementation, we tried to de-couple modules wherever possible, using high-level interfaces to encapsulate the implementations. A good example of this is the notion of a controller model (knob table and measure table), that de-couples the profiler from the runtime, and thus also the controller, allowing us to explore quite different approaches to both control and machine learning without having to change anything in the runtime, or to do any coordination between the implementations.

5.1.5 Treat Application and Hardware as Equals

While our initial system architecture differentiated between application and hardware knobs, we realized that a common abstraction, in the form of a Knob type, could accommodate both hardware and software configuration. This gave us a simple API to instrument adaptable components in the public library interface, that is also used internally, in the runtime, where the support for hardware configuration is implemented. While application code will typically just instantiate knobs, get, and perhaps restrict their values, in the runtime we use the ability to trigger follow-on actions to implement hardware configuration knobs that require system calls whenever their value changes.

5.1.6 Expose Adaptation as a First-Class Object

A key insight for us was the fact that the handling of perturbations such as the loss of a CPU core corresponds to dynamic changes in the configuration space, and so can be implemented using a simple mechanism, where the model that is used by the system's controller is filtered to only include configurations that are compatible with all active perturbations and static constraints. Having this high-level mechanism for manipulating the configuration space led us to expose this functionality in the public library API, through a method on the knob type. Using this method, a programmer can dynamically restrict the values that the controller can use for a

given knob, while leaving the particular choice of the remaining values to the controller. This makes it very easy to implement hybrid imperative-declarative controllers that reconfigure the system automatically by default, based on the intent, but allow the user to make fine-grained decisions about the configuration under specific conditions.

5.2 Key Contributions and Achievements

5.2.1 Key Contributions in Combining Learning and Control

The work on integrating machine learning and control, described in Section 3.3.3, has been accepted for publication in ASPLOS 2018 [35] and in SysML 2018 [36]. The ASPLOS paper has an extensive experimental study demonstrating the benefits of combining learning and control for allocating system resources in mobile and embedded devices. Additional publications include new methods for automatically synthesizing model predictive controllers [33], learning models for non-volatile RAM [17], and modifying the Proteus control system to manage configurations of large scale web infrastructure packages (like MapReduce, HBASE, and Cassandra) [60].

5.2.2 Achievements in ϵ Approximate Control Theory

In a pending publication, we show how to efficiently compute the smart solution (outlined in Section 3.2.2) to an asymmetric problem. In addition, we characterize the decay of a Boolean function, and show how it is related to the influence ratio described above. Further, to connect our model to machine learning, we show how the influence ratio and the decay parameter are related to a key concept in the area of machine learning called *probably approximately correct* (PAC) learning [59]. Finally, we discuss the challenges created by using a stochastic reader modeling the inaccuracy inherent in all real measurements.

5.2.3 Key Contributions of the WMM work

To improve the resilience of parallel programs for embedded systems, we have proposed to use C++11 as the unified programming language model to insulate application programmers from platform-specific details, e.g., compiler optimizations and hardware re-orderings. Furthermore, we have developed a new weak memory model which is resilient to hardware evolution. By using C++11 and WMM in combination, both software and hardware can evolve independently, making the system much more resilient.

Our results (those shown here as well as the more detailed results in: [67, 69, 70]) show that WMM is a practical choice for future hardware developers. Implementation complexity, area, power and performance are all comparable or better than existing commercial memory models. In addition, the simple rigorous definition of WMM and the carefully chosen level of “weakness” mean that a huge variety of future processors can all be implemented without changing the memory model and, therefore, without changing the tricky memory-model-specific code

designed to run on them.

In the course of this research, we have disseminated our results publicly through a variety of peer-reviewed and self-published documents. We have also released our processor designs under open-source software licenses (<https://github.com/csail-csg/riscy-000>) and contributed to the evolving open-source RISC-V ISA [46].

Appendix: Proteus Test Plan

Proteus Phase II Evaluation Test Plan, v1.3

This document describes the tests that we intend to support in the Phase II evaluation of Proteus.

Challenge Problems

Three challenge problems will be used for evaluation. Each is described in its own challenge problem announcement document, hosted on the [github.mit.edu/rice-proteus](https://github.com/mit.edu/rice-proteus) repository.

CP1: radar

- Swift re-implementation of Synthetic Aperture Radar application used in Phase I (that still relies on the C library `tasp-vsip1-core-plus`)

CP2: x264

- Swift wrapper for a version of the mainline x264 source code (not the version from Parsec, which is old and does not match online documentation), customized to allow FAST to control it.

CP3: capsule

- Swift wrapper for a version of the CaPSuLe image matching application, customized to allow FAST to control it.

Test Parameters

Each challenge problem exposes six test parameters to the test harness as shown in Table 8.

Name	Value	Default Value	Description
missionIntent	IntentSpecification	N/A	Changes the intent that is to be met by Proteus. Value can be changed prior to or during test runtime.
availableCores	Integer in {2,4}	4	Change the number of CPU cores on which the application is running. Value can be changed prior to or during test runtime.

availableCoreFreq.	Integer in {600,1200}	1200	Change the operating frequency of the CPU on which the application is running. Value can be changed prior to or during test runtime.
missionLength	Integer in [1,N]	1600	Mission length in number of iterations. Affects the maximum permitted energy consumption. Value can be changed prior to or during test runtime. For RR3, N is 1600 for all CPs.
enforceEnergyLimit	Boolean in {true,false}	false	This parameter makes it possible to test the system with and without the added semantics associated with missionLength. When true, the mission length will be treated as a constraint to be met. When false, the mission length will be treated as a simple upper bound.
sceneImportance	Real in [0.0,1.0]	0.0	Level of detail required of the application output. Affects the minimum permitted quality. Value can be changed prior to or during test runtime.

Table 8: Common Test Parameters

Test Cases

The three test parameters missionIntent, missionLength and sceneImportance are optional, and test cases where they are defined required the remaining ones of these three to be “fixed”, according to the descriptions in the corresponding test cases. For example, in a test case where missionLength is defined, sceneImportance must be set to 0.0, and missionLength must be set to a specific value at initialization, be left unperturbed throughout the test, and the parame-

ter enforceEnergyLimit (that enabled the intended semantics of missionLength) must be set to false. Thus, we divide the test suite into three parts that each test one of these three test parameters. Because the evaluator will perform statistical tests of each performer's system, most of the test cases in this document are actually *sets* of test cases, with parameters chosen from the given ranges. Though the test cases are designed to be application-independent, their parameters are not. For example, the range of constraint goals that are achievable are different for each application.

Notation

In the "Type" column in the tables in this section, "I" stands for an "interesting" case that showcases/validates normal system behavior, and "C" stands for a "corner case", where the system executes outside of intended operating conditions. We use the following terminology to describe the components of an intent specification's intent section, which has the general form: "O(F) such that $L = R$ ", where:

- O is the optimization type
- F is the objective function
- L is the constraint measure
- R is the constraint goal

Expected Verdict

The "Expected verdict" column lists the verdicts that are expected from the test, as defined by the verdict expression (candidate version will be delivered by RR3). Whenever a property of a test case should be validated by manual inspection, an additional comment is given in this column.

Settling Time Interval

The challenge problems are designed to validate that Proteus behaves as expected during execution, interrupted by a few perturbations. This is primarily a matter of test performance, since running a test with many perturbations would require running applications for many iterations, which would result in very long test times. To allow the tests to focus on correct steady-state behavior (both before and after perturbations), we allow the system to settle over a number of iterations, by eliminating that number of iterations from the data used to compute verdicts. We call this number of iterations (that for simplicity is the upper bound of iterations required by each application) the *settling time interval* (STI), and set it to 200 for all tests, aside from those where the STI is explicitly stated in the "Parameters" column. For all test cases where a perturbation is sent to Proteus during execution, the perturbation must not occur until after at least STI iterations have completed following initialization or a previous perturbation.

Achievable Measure Values

Proteus adapts to perturbations by choosing from a set of available configurations. Each of these configurations is associated with an average value, that is computed by profiling prior to executing the tests. The set of these averages for all available configurations is an estimate of

the measure values that Proteus can achieve by switching into the corresponding configuration. Given a measure m we call this set the *value range* of the measure, and denote it by V_m . The *dynamic range* of m is defined as $D_m = [\min(V_m), \max(V_m)]$. We let $D_{m,k=v}$ denote the dynamic range of m , restricted to configurations where the value of knob k is v .

The following table defines D_m for every measure that occurs in the test cases.

		D_m		
Measure	Description	CP1	CP2	CP3
performance	Iterations per second	[34.7,332.0]	[3.13,336.6]	[0.096,0.43]
quality	Application-specific	[0.53,3.2]	[37228.7,63116.7]	[878.9, 110394.1]
energyDelta	Microjoules / iteration	[64651.6,189113.9]	[70583.6,977282.2]	[7341771.4,31715946.1]

Table 9: Definition of D_m

Optimization problem per challenge problem

The tests are parameterized by an optimization problem P_C for each challenge problem C , consisting of an optimization type P_C^T (one of min and max), and an objective function P_C^F , as specified by this table. Some test cases uses the function `invert()` that changes an optimization type min to max, and max to min.

Challenge Problem C	Optimization type P_C^T	Objective function P_C^F
CP1	max	quality
CP2	max	quality
CP3	min	powerConsumption

Table 10: Optimization Problem Per Challenge Problem

Test Cases for the missionIntent Parameter

#	Type	Test case	How to generate it	Expected verdict
---	------	-----------	--------------------	------------------

1	I	Jump in constraint goal	Initialize the missionIntent parameter to: P_C such that performance = G_1 and perturb it to P_C such that performance = G_2 where $G_1, G_2 \in D_{\text{performance}}$ and $ G_1 - G_2 > 0.2 * D_{\text{performance}} $	PASS
2	I	Change between constraint measures	Initialize the missionIntent parameter to: P_C such that performance = G_1 and perturb it to P_C such that latency = G_2 where $G_1 \in D_{\text{performance}}$ and $G_2 = 1/G_1$	PASS
3	I	Inverting the optimization type (min to max, or max to min) and negate the objective function	Initialize the missionIntent parameter to: $P_C^T(P_C^F)$ such that performance = G_1 and perturb it to $\text{invert}(P_C^T)(-P_C^F)$ such that performance = G_1 where $G_1 \in D_{\text{performance}}$	PASS
4	I	Negating the objective function	Initialize the missionIntent parameter to: $P_C^T(P_C^F)$ such that performance = G_1 and perturb it to $P_C^T(-P_C^F)$ such that performance = G_1 where $G_1 \in D_{\text{performance}}$	PASS <i>An change in P_C^F should occur after the perturbation.</i>
5	I	Multiple jumps in constraint goal	Initialize the missionIntent parameter to: P_C such that performance = G_1 then perturb it to P_C such that performance = G_2 then perturb it to P_C such that performance = G_3 where $G_1, G_2, G_3 \in D_{\text{performance}}$, $ G_1 - G_2 > 0.2 * D_{\text{performance}} $, and $ G_2 - G_3 > 0.2 * D_{\text{performance}} $	PASS

6	C	Two consecutive jumps in constraint goal without enough time to meet the intent	For this test case, let $STI = 0$ (we denote the original STI as STI_0 below) and $missionLength = T_1 + T_2 + T_3$, where $T_1, T_2, T_3 \in [STI_0 / 8, STI_0 / 2]$. Initialize the missionIntent parameter to: P_C such that performance = G_1 then, after T_1 iterations, perturb it to P_C such that performance = G_2 then, after T_2 iterations, perturb it to P_C such that performance = G_3 where $G_1 \in \{\min(D_{performance}), \max(D_{performance})\}$, $G_1 - G_2 > 0.8 * D_{performance}$, $G_2 - G_3 > 0.8 * D_{performance}$.	FAIL
7	C	Constraint goal not achievable	Initialize the missionIntent parameter to: P_C such that performance = G_1 where $G_1 = \max(D_{performance}) + D_{performance}$	FAIL <i>Value of the performance measure should be close to $\max(D_{performance})$.</i>

Table 11: Test Cases for the missionIntent Parameter

Test Cases for the missionLength Parameter

Figure 31 and paragraphs explain how an *achievable* missionLength can be randomized for the purpose of testing, based on information about energy consumption per iteration (energy deltas $e\Delta$) that are found in the controller model.

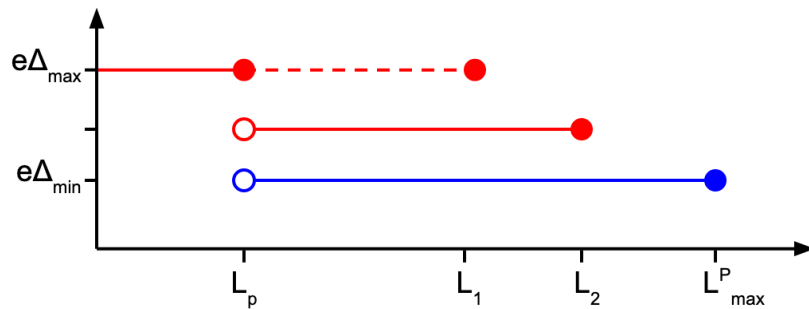


Figure 31: Derivation of achievable mission length

We assume that, before the mission starts, that the base equips the system with sufficient energy resources to complete $missionLength = L_1$ iterations *at the most energy-inefficient configuration* $e\Delta_{max}$, that is **energyLimit** = $e\Delta_{max} * L_1$.

Approved for Public Release; Distribution Unlimited.

At some iteration $L_p < L_1$ during the mission, the base decides to extend the missionLength to some $L_2 > L_1$. At L_p , we can conservatively estimate the amount of energy that has been spent as $e\Delta_{\max} * L_p$, and thus the energy remaining to achieve the remainder of the new missionLength is:

$$\begin{aligned} e_{\text{remaining}} &= \text{energyLimit} - (e\Delta_{\max} * L_p) \\ &= (e\Delta_{\max} * L_1) - (e\Delta_{\max} * L_p) \\ &= e\Delta_{\max} * (L_1 - L_p) \end{aligned}$$

Since the maximum number of iterations is always achieved using the most energy-efficient configuration $e\Delta_{\min}$, the maximum achievable missionLength $L_{\max}^p$ given a change at L_p is:

$$\begin{aligned} L_{\max}^p &= L_p + e_{\text{remaining}} / e\Delta_{\min} \\ &= L_p + e\Delta_{\max} * (L_1 - L_p) / e\Delta_{\min} \\ &= L_p + (e\Delta_{\max} / e\Delta_{\min}) * (L_1 - L_p) \end{aligned}$$

Based on this upper bound, we can define a test case for mission length as follows.

The test for missionLength depends on a performance goal G_C , for each challenge problem C , and a corresponding maximal post-perturbation missionLength L_C (assuming the perturbation happens at iteration 800) that have been manually selected to illustrate the effect of this parameter, in the sense that the energyDelta and quality measures after the perturbation decrease (to the extent that this is possible, while satisfying the performance goal).

Challenge problem C	CP1	CP2	CP3
G_C	150.0	50.0	0.3
L_C	5445	15437	4739

Table 12: Goal and Maximal Post-Perturbation Mission Length Per Challenge Problem

#	Type	Test case	How to generate it	Expected verdict
8	I	Increase in missionLength	Set the enforceEnergyLimit parameter to true for this test case. Initialize the missionLength parameter to 1600 and set the missionIntent parameter to: max(quality) such that performance = G_C then at iteration 800 perturb missionLength to L_C	PASS <i>Validate manually that the energyDelta and quality measures decrease (on average) after the perturbation.</i>

Table 13: Test Cases for the missionLength Parameter

Test Cases for the sceneImportance Parameter

The test for sceneImportance depends on a performance goal G_C , corresponding to each challenge problem C, that has been manually selected to illustrate the effect of this parameter, in the sense that the quality measure after the perturbation increases (to the extent that this is possible, while satisfying the performance goal).

Challenge problem C	CP1	CP2	CP3
G_C	100.0	41.0	0.2

Table 14: sceneImportance Test Case Performance Goals Per Challenge Problem

#	Type	Test case	How to generate it	Expected verdict
9	I	Increase from low to high importance	Initialize the sceneImportance parameter to: 0.0 and the missionIntent parameter to: $\min(\text{powerConsumption})$ such that $\text{performance} = G_C$ and then perturb the sceneImportance parameter to: S where $S \in [0.8, 1.0]$	<i>Validate manually (e.g. by inspecting a plot of quality versus time) that the quality measure after the perturbation increases.</i>

Table 15: Test Cases for the sceneImportance Parameter

Test Case for the availableCores Parameter

This test case demonstrates the ability of Proteus to keep meeting a goal G_C , despite a loss of available cores during the course of execution, as long as that goal is still achievable after the perturbation (when the utilizedCores knob is set to 2). The range of achievable goals G_C , in configurations where the utilizedCores knob is set to 2, is approximated by $D^C_{\text{performance,utilizedCores}=2}$, given in this table for each challenge problem C:

Challenge problem C	CP1	CP2	CP3
$D^C_{\text{performance,utilizedCores}=2}$	[34.7,311.0]	[3.2,170.4]	[0.096,0.39]

Table 16: Bounds on Achievable Performance Goals When utilizedCores is 2

When executing this test case, the remaining test parameters should be set to their default values.

#	Type	Test case	How to generate it	Expected verdict
10	I	Disable half the cores	Initialize the availableCores parameter to: 4 and the missionIntent parameter to: P_C such that $\text{performance} = G_C$ where $G_C = \max(D^C_{\text{performance,utilizedCores}=2})$ and then perturb the availableCores parameter to: 2	PASS

Table 17: Test Case for availableCores Parameter

Approved for Public Release; Distribution Unlimited.

Test Case for the availableCoreFrequency Parameter

This test case demonstrates the ability of Proteus to keep meeting a goal G_C , despite a loss of available core frequencies during the course of execution, as long as that goal is still achievable after the perturbation (when the utilizedCoreFrequency knob is set to 600). The range of achievable goals G_C , in configurations where the utilizedCoreFrequency knob is set to 600, is approximated by $D^C_{\text{performance,utilizedCoreFrequency}=600}$, given in this table for each challenge problem C:

Challenge problem C	CP1	CP2	CP3
$D^C_{\text{performance,utilizedCoreFrequency}=600}$	[34.7,186.7]	[3.2,181.2]	[0.096,0.22]

Table 18: Bounds on Achievable Performance Goals Where utilizedCoreFrequency is 600

When executing this test case, the remaining test parameters should be set to their default values.

#	Type	Test case	How to generate it	Expected verdict
11	I	Limit the core frequency	Initialize the availableCoreFrequency parameter to: 1200 and the missionIntent parameter to: P_C such that performance = G where G = $\max(D^C_{\text{performance,utilizedCoreFrequency}=600})$, that is, the maximal performance achievable with a utilizedCoreFrequency of 600MHz, and then perturb the availableCoreFrequency parameter to: 600	PASS

Table 19: Test Case for the availableCoreFrequency Parameter

Proteus Phase III Evaluation Test Plan, v0.7

This document describes a test plan consisting of a set of tests whose purpose is to demonstrate and validate that applications running under the control of Proteus are capable of meeting dynamic changes in the operating environment and the mission intent when running on any hardware platform supported by the Proteus runtime.

Challenge Problems

Three challenge problems will be used for evaluation. Each is described in its own challenge problem announcement document, hosted on the git.isis.vanderbilt.edu/SwRI/rice/rice-proteus repository.

- CP1: [flightTestScenario7](#)
 - A FAST application that enables dynamic control of radio link bandwidth allocation by Proteus through the SwRI OrionDB database.
- CP2: [x264](#)
 - FAST wrapper for a version of the mainline x264 source code (not the version from Parsec, which is old and does not match online documentation), customized to allow Proteus to control it.
- CP3: [capsule](#)
 - FAST wrapper for a version of the CaPSuLe image matching application, customized to allow Proteus to control it.

This document only describes the test plan for CP2 and CP3. The test plan for CP1 is to be decided together with SwRI. The hardware platform chosen for the evaluation of CP2 and CP3 is the Xilinx ZCU 102.

Test Parameters

Each challenge problem exposes three test parameters to the SwRI test harness.

Name	Value	Default Value	Description
missionIntent	IntentSpecification	N/A	Changes the intent that is to be met by Proteus. Value can be changed prior to or during test runtime.
availableCores	Integer in {2,4}	4	Change the number of CPU cores on which the application is running. Value can be changed prior to or during test runtime.

Approved for Public Release; Distribution Unlimited.

Name	Value	Default Value	Description
availableCoreFrequency	Integer in {600,1200}	1200	Change the operating frequency of the CPU on which the application is running. Value can be changed prior to or during test runtime.

Table 20: Common Test Parameters

Test Cases

The test cases described in this document specify how values for the missionIntent test parameter can be randomly generated to enable effective evaluation of Proteus. Because SwRI will perform statistical tests of each performer’s system, most of the test cases in this document are actually *sets* of test cases, with parameters chosen from the given ranges. Though the test cases are designed to be application-independent, their parameters are not. For example, the range of constraint goals that are achievable are different for each application.

Notation

In the “Type” column in the tables in this section, “I” stands for an “interesting” case that showcases/validates normal system behavior, and “C” stands for a “corner case”, where the system executes outside of intended operating conditions. We use the following terminology to describe the components of an intent specification’s intent section, which has the general form: “T(F) such that $L = R$ ”, where:

- T is the optimization type
- F is the objective function
- L is the constraint measure
- R is the constraint goal

Expected Verdict The “Expected verdict” column lists the verdicts that are expected from the test, as defined by the verdict expression (candidate version will be delivered by RR3). Whenever a property of a test case should be validated by manual inspection, an additional comment is given in this column.

Settling Time Interval The challenge problems are designed to validate that Proteus behaves as expected during execution, interrupted by a few perturbations. This is primarily a matter of test performance, since running a test with many perturbations would require running applications for many iterations, which would result in very long test times. To allow the tests to focus on correct steady-state behavior (both before and after perturbations), we allow the system to settle over a number of iterations, by eliminating that number of iterations from the data used

to compute verdicts. We call this number of iterations (that for simplicity is the upper bound of iterations required by each application) the *settling time interval* (STI), and set it to 200 for all tests, aside from those where the STI is explicitly stated in the “Parameters” column. For all test cases where a perturbation is sent to Proteus during execution, the perturbation must not occur until after at least STI iterations have completed following initialization or a previous perturbation.

Models and Achievable Measure Values Proteus adapts to perturbations by choosing from a set of available configurations. Each of these configurations is associated with an average value, that is computed by profiling prior to executing the tests. The set of these averages for all available configurations is an estimate of the measure values that Proteus can achieve by switching into the corresponding configuration. The association of configuration to its expected measure values is called a *model*. The models M_c (each consisting of two CSV files, a “.knobtable” and a “.measuretable”, joined by a shared configuration “id” column) for each challenge problem c , are available in the rice-proteus repository:

- M_{CP2} : [knobtable](#), [measuretable](#)
- M_{CP3} : [knobtable](#), [measuretable](#)

Based on a model, the set of all measure values A_c for the challenge problem c achievable by Proteus can be defined as the measure values of each possible *schedule*, where a schedule is a sequence of configurations. Given a model M and a schedule S of configurations from M , the expected vector of average measure values of S is $E_s = t_n M^n$, where t_n is the total time allocated to configuration c in S , and M^n is the expected vector of measure values for configuration n in M . The definition of E_s gives us a way to generate achievable measure values, based on a model, by simply picking a random schedule of configurations from the model, and computing its expected measure values.

Because the computed expected measure values are based on a model, it is possible that they may not be achievable at all times during an actual execution¹. To make this less likely, the goals can be made more conservative by adding a “padding”. As illustrated in Figure 32, this can be seen as adding a vector to the generated goals in the space of measure values, to bring the goals away from the convex hull of this space. Given the center of mass C of the space, a vector G of expected measure values of a given randomized schedule, and a scalar padding (equal to 0.1 in our tests) the padded goals are equal to $G + P \cdot (C - G)$.

To demonstrate the adaptive behavior of Proteus, when the randomized goals are converted into constraints, each measure’s goal is equipped with a relation (one of “ $\leq$ ” or “ $\geq$ ”) that “points away from the center of mass”. Since successive goals are generated to be at a given distance away from one another, this tends to produce constraints that force Proteus to switch configurations in each measure after each perturbation.

A concrete example of this approach to generating valid achievable combinations of measure values (for CP2 and CP3) is available [here](#) (function `gen_goals()` and its utility `pad_goals_and_add_relations()`).

¹Such a discrepancy can be caused by a dependence of a certain measure on the current input. For example, this happens in CP2, where the performance in a given configuration decreases during execution (when x264 is encoding default video stream for Phase III).

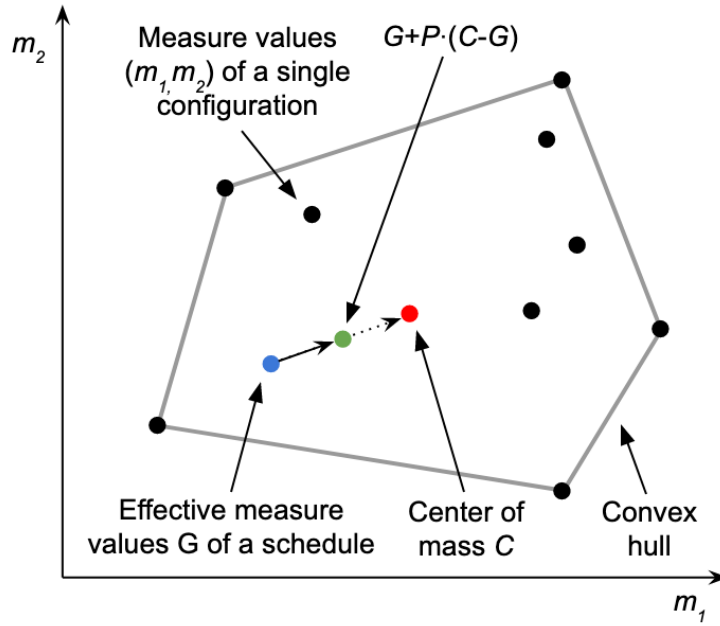


Figure 32: Generating achievable goals for a constraint with two measures m_1 and m_2 by randomizing schedules

To simplify the definition of test cases based on these definitions, we let E_M^W be the set of expected measure values E_S of all possible schedules S of length W with configurations taken from the model M . The window size and model are fixed for each challenge problem as follows, and we will thus refer to E_M^W simply as E_c where c is a challenge problem.

Table 21: Window Sizes for CP2 and CP3

	CP2	CP3
Window size	40	20

Given a measure m we call the set of values the measure can achieve across all the configurations in the model the *value range* of the measure, and denote it by V_m . The *dynamic range* of m is defined as $D_m = [\min(V_m), \max(V_m)]$. We let $D_{m,k=v}$ denote the dynamic range of m , restricted to configurations where the value of knob k is v .

The following table defines D_m for every measure that occurs in the test cases.

Table 22: Definition of D_m

		D_m	
Measure	Description	CP2	CP3
performance	Iterations per second	[0.628, 6.36]	[0.131, 0.444]
quality	Application-specific	[57085.4, 59259.5]	[878.9, 219880.43]
energyDelta	Microjoules / iteration	[560281.8, 5149814.5]	[7006588.1, 23482102.5]

Optimization problem per challenge problem Besides the constraint, the tests are also parameterized by an optimization problem $P_C = T_C(F_C)$ for each challenge problem c , consisting of an optimization type T_C (one of min and max), and an objective function F_C , as specified by this table. Some test cases uses the function invert() that changes an optimization type min to max, and max to min.

Optimization Problem $P_C = T_C(F_C)$		
Challenge Problem c	Optimization type T_C	Objective function F_c
CP2	max	quality
CP3	min	powerConsumption

Table 23: Optimization Problem per Challenge Problem

b

Test Cases for the missionIntent Parameter

Table 24 below lists the test cases for the missionIntent parameter.

#	Type	Test case	How to generate it	Expected verdict
1	I	Jump in constraint goal	Initialize the missionIntent parameter to: P_C such that performance = G_1 and perturb it to P_C such that performance = G_2 where $G_1, G_2 \in D_{\text{performance}}$ and $ G_1 - G_2 > 0.2 * D_{\text{performance}} $	PASS
2	I	Change between constraint measures	Initialize the missionIntent parameter to: P_C such that performance = G_1 and perturb it to P_C such that latency = G_2 where $G_1 \in D_{\text{performance}}$ and $G_2 = 1/G_1$	PASS
3	I	Inverting the optimization type (min to max, or max to min) and negate the objective function	Initialize the missionIntent parameter to: $T_C(F_C)$ such that performance = G_1 and perturb it to $\text{invert}(T_C)(-F_C)$ such that performance = G_1 where $G_1 \in D_{\text{performance}}$	PASS

#	Type	Test case	How to generate it	Expected verdict
4	I	Negating the objective function	Initialize the missionIntent parameter to: $T_C(F_C)$ such that performance = G_1 and perturb it to $T_C(-F_C)$ such that performance = G_1 where $G_1 \in D_{\text{performance}}$	PASS <i>A change in F_C should occur after the perturbation.</i>
5	I	Multiple jumps in constraint goal	Initialize the missionIntent parameter to: P_C such that $K(G_1)$ then perturb it to P_C such that $K(G_2)$ then perturb it to P_C such that $K(G_3)$ where for $k \in \{1,2,3\}$, $G_k \in E_C$ the constraint $K(G_k)$ is a conjunction of non-strict inequalities such that $\forall m \in Z. \forall k \in \{2,3\}. (D_m = 0 \text{ or } (K(G_k)[m] - K(G_{k-1})[m] / D_m) > 0.2)$, where Z is the set of measure names in $K(G_k)$, and $K(G_k)[m]$ is the constraint value of measure m in the constraint $K(G_m)$. For example, $K(G_k)$ could be “performance ≥ 20.0 and quality ≥ 52000.0 ”, where $(20.0, 52000.0) \in E_C$.	PASS
6	C	Constraint goal not achievable	Initialize the missionIntent parameter to: P_C such that performance = G_1 where $G_1 = \max(D_{\text{performance}}) + D_{\text{performance}} $	INVALID <i>Value of the performance measure should be close to $\max(D_{\text{performance}})$.</i>

Table 24: Test Cases for the missionIntent Parameter

Test Case for the availableCores Parameter

This test case, detailed in Table 26, demonstrates the ability of Proteus to keep meeting a goal G_C , despite a loss of available cores during the course of execution, as long as that goal is still achievable after the perturbation (when the utilizedCores knob is set to 2). The range of achievable goals G_C , in configurations where the utilizedCores knob is set to 2, is approximated by $D_{\text{performance}, \text{utilizedCores}=2}^C$, given for each challenge problem C in Table 25.

Challenge problem C	CP2	CP3
$D^C_{\text{performance,utilizedCores}=2}$	[0.628,3.067]	[0.131,0.407]

Table 25: Bounds on Achievable Performance Goals When utilizedCores is 2

When executing this test case, the remaining test parameters should be set to their default values.

#	Type	Test case	How to generate it	Expected verdict
7	I	Disable half the cores	Initialize the availableCores parameter to: 4 and the missionIntent parameter to: P_C such that performance = G_C where $G_C = \max(D^C_{\text{performance,utilizedCores}=2})$ and then perturb the availableCores parameter to: 2	PASS

Table 26: Test Case for the availableCores Parameter

Test Case for the availableCoreFrequency Parameter

This test case, detailed in Table 28, demonstrates the ability of Proteus to keep meeting a goal G_C , despite a loss of available core frequencies during the course of execution, as long as that goal is still achievable after the perturbation (when the utilizedCoreFrequency knob is set to 600). The range of achievable goals G_C , in configurations where the utilizedCoreFrequency knob is set to 600, is approximated by $D^C_{\text{performance,utilizedCoreFrequency}=600}$, given for each challenge problem C in Table 27.

Challenge problem C	CP2	CP3
$D^C_{\text{performance,utilizedCoreFrequency}=600}$	[0.628,6.26]	[0.131,0.443]

Table 27: Bounds on Performance Goals When utilizedCoreFrequency is 600

When executing this test case, the remaining test parameters should be set to their default values.

#	Type	Test case	How to generate it	Expected verdict
8	I	Limit the core frequency	Initialize the availableCoreFrequency parameter to: 1200 and the missionIntent parameter to: P_C such that performance = G where $G = \max(D^C_{\text{performance,utilizedCoreFrequency=600}})$, that is, the maximal performance achievable with a utilizedCoreFrequency of 600MHz, and then perturb the availableCoreFrequency parameter to: 600	PASS

Table 28: Test Case for the availableCoreFrequency Parameter

Appendix: Challenge Problem SAR

Rice Proteus Phase I/II Challenge Problem 1: Synthetic Aperture Radar (radar)

Overview

The mission scenario is unchanged from Phase I. The context of the challenge problems is an embedded system, *e.g.* a UAV equipped with various sensors that is conducting a designated mission. The UAV must process the incoming data from the sensors and report its analysis back to base. The available resources may fluctuate, *e.g.*, battery level, available uplink bandwidth, *etc.* The system will try to adapt to these changes by utilizing online control and machine learning supported by off-line machine learning in order to maintain the quality requirements posed by the base through our novel programming language FAST that includes application intent specifications in addition to conventional code. The aforementioned tasks are carried out by a dedicated on-board system (independent from the avionics control) running the corresponding intent-enriched application.

Compared to Phase I, two additional aspects of this scenario are reflected in the challenge problem. The effect of mission length on achievable accuracy is captured by the additional test parameter `missionLength`. The effect of priority (the "importance" of the scene currently processed by the application) on the quality of the the output is captured by the additional test parameter `sceneImportance`. This challenge problem will focus on demonstrating that Proteus is able to meet intents by adapting combinations of system configuration knobs and application knobs, which was not demonstrated in Phase I.

Test Data

The radar application will use synthetic test data generated by the application itself.

Test Parameters

Five test parameters will be exposed for CP1. The first is `missionIntent` that corresponds to the performance constraint that Proteus had to meet in Phase I. We are currently discussing extensions of this capability, based on the flexibility of the syntax of the FAST Intent Specification Language, and the underlying controller and optimizer required to support that flexibility, that will affect the type of this parameter. The second and third test parameters (`availableCores` and `availableCoreFrequency`) constrain the permissible ranges of system configuration knobs, analogous to those used in Phase I evaluation, but adapted to the target Xilinx ZCU platform. The fourth and fifth test parameters (`missionLength` and `sceneImportance`), that are new for Phase II, will contribute to making the problem more challenging, by increasing the dimension and size of the state space, since Proteus will react to changes in these parameters by adapting both system and application knobs.

Name	Value	Description
missionIntent	IntentSpecification	Changes the intent that is to be met by Proteus. Value can be changed prior to or during test runtime. Instructions for generating interesting values for test are to be detailed.
availableCores	Integer in {2,4}	Change the number of CPU cores on which the application is running. Value can be changed prior to or during test runtime.
availableCoreFrequency	Integer in {600,1200}	Change the operating frequency of the CPU on which the application is running. Value can be changed prior to or during test runtime.
missionLength	Integer in [1,1600]	Total number of input units to process. Affects the maximum permitted energy usage. Value can be changed prior to or during test runtime.
enforceEnergyLimit	Boolean in {true, false}	When true, missionLength is treated as a constraint to be met. When false, missionLength is treated as a simple upper bound on the number of iterations the application should execute.

Name	Value	Description
sceneImportance	Real in [0,1]	Importance of the current processing. Determines the degree to which low-quality, low-resource configurations should be avoided, in favor of high-quality, high-resource configurations. When sceneImportance is 0.0, all configurations are considered by the system. When sceneImportance is 1.0, only the highest-quality configurations are considered.

Table 29: Challenge Problem SAR Test Parameters

We aim to increase the variability of availableCores and availableCoreFrequency test parameters by introducing additional possible values by the code freeze.

The syntax of the missionIntent test parameter is given by the [IntentSpecification](#) schema. It includes a set of *knob* declarations, which represent the different configurations that the system can enter to adapt to meet the mission intent, or to adapt to perturbations. For the purpose of Phase II evaluation, Proteus (and its emulation component) supports setting the knobs of both the application and the system to the values listed in the table below. The example in the section "Intent Specification and Evaluation Metrics" at the end of this document shows how the possible knob values are represented in an intent specification.

Name	Value	Description
cdr	Integer in {8,16}	Coarse decimation ratio. Increasing this knob should increase the performance measure, in exchange for a decrease in the quality measure.
fdr	Integer in {8,16}	Fine decimation ratio. Increasing this knob should increase the performance measure, in exchange for a decrease in the quality measure.
numberOfRanges	Integer in {1024,2048}	Number of synthetic radar ranges. Increasing this knob should increase the performance measure, in exchange for a decrease in the quality measure.

Name	Value	Description
numberOfBeams	Integer in {8,16}	Number of synthetic radar beams. Increasing this knob should increase the performance measure, in exchange for a decrease in the quality measure.
utilizedCores	Integer in {2,4}	Number of CPU cores that the application should distribute its computation over. Increasing this knob should increase the performance measure, in exchange for a decrease in the powerConsumption/energy-Delta measures. See side note below.
utilizedCoreFrequency	Integer in {600,1200}	The CPU clock frequency that the system should be configured to. Increasing this knob should increase the performance measure, in exchange for a decrease in the powerConsumption/energyDelta measures. See side note below.

Table 30: Challenge Problem SAR Knobs

Side note about the utilizedCores and utilizedCoreFrequency knobs: The tradeoff exposed by these knobs is counter intuitive since, generally, higher clock frequencies require a higher voltage, and using more cores requires powering a bigger circuit, and thus should require more energy. However, due to the unthrottled nature of the Phase II CPs (in contrast to actual sensor inputs, that may be read at a fixed rate), the processing of iterations can happen at an arbitrarily high rate. This means that it can be optimal, from an energy consumption standpoint, to process all iterations at the highest possible frequency / most cores, allowing the system to complete the processing as soon as possible, since, at lower frequencies / less cores, energy is spent not only on powering the CPU, but also other system components (accounted for in the energy-derived measures). Consequently, whenever the intent specification mandates minimizing energy (or powerConsumption or energyDelta), Proteus will only use configurations that include low frequencies / few cores when it has no other choice (such as when the availableCoreFrequency test parameter specifies that a higher frequency is not available).

Test Procedure

In Phase I, the oracle compared Proteus to a clairvoyant non-adaptive system. It was constructed by running the system in ten Pareto-optimal configurations, and choosing the best one w.r.t. the intent. This served well to showcase the benefit of adaptation. However, it was not clear how much insight it gave into how good Proteus is at adaptation.

To improve on this, we have developed an alternative oracle, whose behavior illustrated in Figure 33.

In this plot, the vertical axis represents the constraint value (in this case, the window performance, that is, the number of inputs processed per second). The green line represents the goal

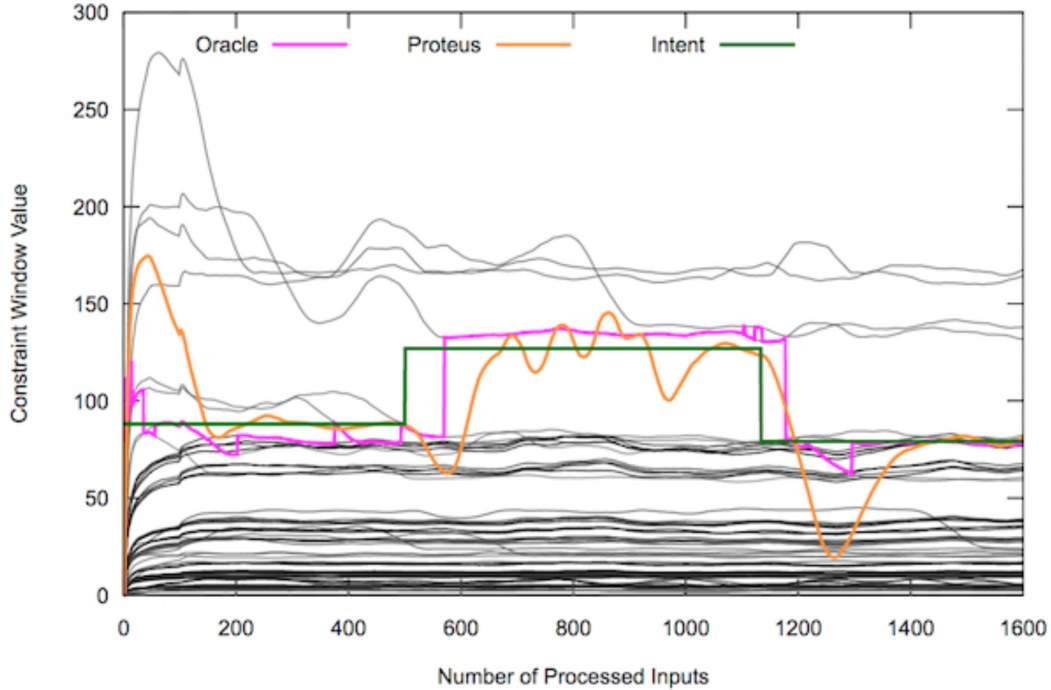


Figure 33: Behavior of the Phase III oracle

stated in the intent specification. The orange line represents the window average of the observed performance of Proteus, executing with adaptation enabled. The pink line represents the performance of the Oracle, which is constructed from logs gathered during a set of executions, each corresponding to a fixed configuration. Each of the grayscale lines in the plot corresponds to one such execution. Note that, although the system is running in a fixed configuration for each of these executions, the observed measure (performance in this case) values fluctuate. This is due to several factors, including variability in system resources (e.g. operating system process scheduling) and the possibly input-dependent nature of the application (a frame of a video may take a different amount of time/resources to process).

The oracle is defined in terms of several parameters:

- **GLOBAL_CONSTRAINT_ERROR_THRESHOLD (GCET)** A threshold on the cumulative relative absolute error (versus the constraint goal), below which an execution is considered to have met the constraint.
- **GLOBAL_OBJECTIVE_FUNCTION_ERROR_THRESHOLD (GOFET)** A threshold on the objective function error (versus the oracle), below which a Proteus execution is considered to be sufficiently close to the optimum.
- **LOCAL_ERROR_THRESHOLD (LET)** A threshold on the relative window error (versus the constraint goal), below which a configuration is considered to meet the *equality* constraint (at a given input).
- **WINDOW_SIZE (WS)** The number of inputs over which window averages are computed.
- **STABILIZATION_PERIOD (SP)** The number of inputs after a perturbation (including the start of the execution) that are excluded from the verdict computation. This is equal to 200 for all CPs.

The values of the first three of the above parameters are specific to each challenge problem:

Approved for Public Release; Distribution Unlimited.

Parameter	CP1	CP2	CP3
GLOBAL_CONSTRAINT_ERROR_THRESHOLD	0.15	0.4	0.05
GLOBAL_OBJECTIVE_FUNCTION_ERROR_THRESHOLD	0.6	0.5	0.1
LOCAL_ERROR_THRESHOLD	0.15	0.4	0.05

Table 31: Challenge Problem SAR Verdict Expression Parameters

The oracle is then used to define a verdict expression.
In the sections below:

- MARE stands for mean absolute relative error.
- FCE stands for fixed-configuration execution.

Oracle

The oracle attempts to meet the *equality* constraint *at each input processed during the execution*. It is meant to resemble Proteus by adapting to changes in the `missionIntent`, unlike the Phase I oracle, which represented an idealized non-adaptive system. The Phase II oracle is constructed as follows. For each input, the oracle has the behavior (the values and statistics) of one of the FCEs at that input. If, for this input, none of the FCEs meet the constraint (their MARE exceeds `LOCAL_ERROR_THRESHOLD`), then select the behavior of the FCE with the smallest MARE. Otherwise, select the one of those FCEs that meet the constraint with the best objective function, that is, highest or lowest, depending on the optimization type (min or max) given in the intent specification at the corresponding input.

Verdict Expression

The verdict expression is defined based on the oracle as illustrated in Figure 34, where:

- e_p and e_o stand for the combined mean relative errors of the Proteus and oracle executions, respectively.
- The relative objective function oracle advantage (ROFOA), which is a measure of how much better the oracle's objective function is compared to that of Proteus (set to 0 when Proteus surpasses the oracle), is defined as follows. Let f_p and f_o be the normalized objective functions of the Proteus and oracle executions, respectively (if the optimization type is min, then the original objective function is negated to give the normalized objective function; if it is max, the normalized objective function is equal to the original objective function). Further let $f_{o,min}$ and $f_{o,max}$ be the minimal and maximal normalized objective functions counted across all oracle executions. Then, define ROFOA as 0 if $f_o \leq f_p$ and $(f_o - f_p) / (f_{o,max} - f_{o,min})$ otherwise.

Interface to the Test Harness (API)

In Phase II, Proteus will be communicating with the Test Harness over RESTful APIs in two ways:

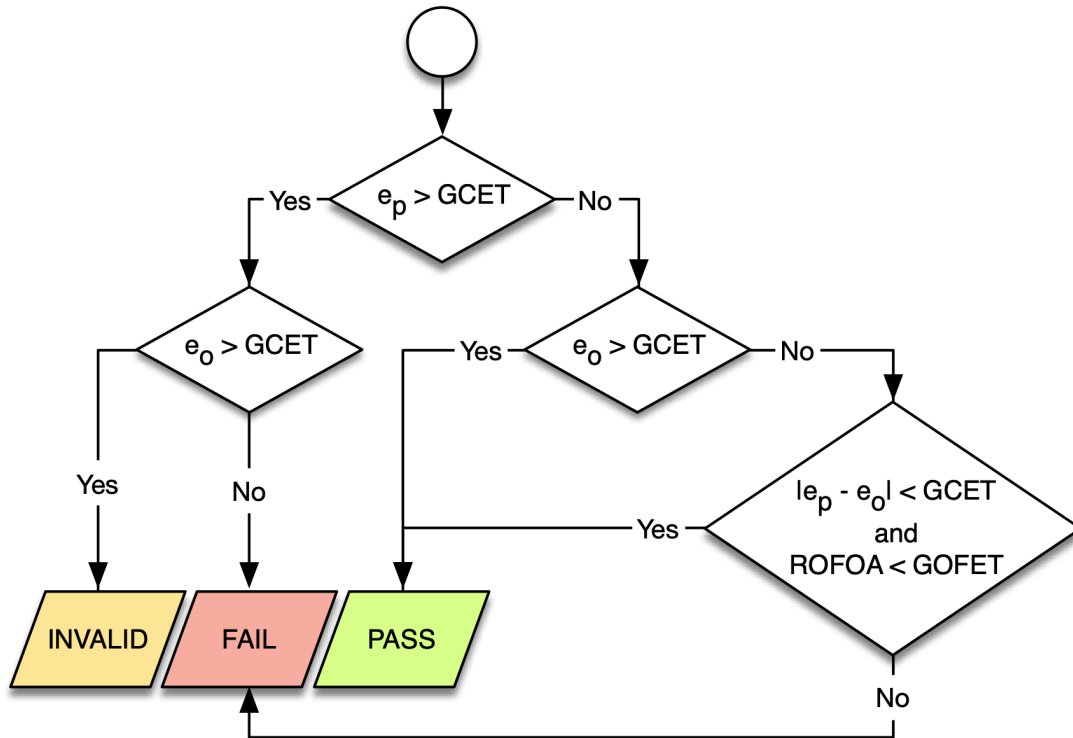


Figure 34: Phase III verdict expression

- a **Server** (part of the Proteus runtime) that the Test Harness can call to query and perturb the state of a Challenge Problem application;
- a **Client** that is used to call REST endpoints exposed by the Test Harness.

1. **TH**: Starts the SUT (a particular CP application).
2. **SUT**: Starts the Proteus REST server.
3. **SUT**: Posts to TH/ready, and receives initialization message in the response. An example of the JSON initialization message is given in the section Initial Parameters below.
4. **SUT**: Initializes the Proteus runtime with respect to the response from TH/ready.
5. **SUT**: Posts to TH/initialized.
6. Loop until end of input stream (**either of these could happen at any time**):
 - TH drives the test one "step" forward (**by posting to either of these**):
 - **TH**: Posts to SUT/process with a message containing the number of inputs to process.
 - **TH**: Posts to SUT/perturb with a message containing the test parameters. The **SUT** responds synchronously with a message containing the status after the perturbation has been applied.
 - **SUT**: Posts to TH/status after the processing of each input.
 - **TH**: Gets from SUT/alive to obtain SUT liveness information.
 - **TH**: Gets from SUT/query to obtain SUT status.
7. **SUT**: Posts to TH/done to signal that all the inputs in the input stream have been processed.

Server

As a **Server**, Proteus will expose a RESTful API with four endpoints. As recommended by Lincoln Laboratory (LL), we use the formal specification standard [Swagger](#) to document the API. The Proteus REST API ([RestServer.yaml](#)) is available as part of the FAST source code. The main constituents of the specification are the paths, which specify the REST endpoints of the API, and the definitions, which specify the types used for input and output parameters. The paths are as follows:

- **/alive**: the TH issues a request to this endpoint to query the liveness of the SUT.
- **/process**: the TH issues a request to this endpoint to have the SUT process a number of inputs as specified in the request JSON payload.
- **/perturb**: the TH issues a request to this endpoint with a JSON message containing appropriate entries defining the perturbations to which the SUT must respond. An example perturbation may set the performance goal and the mission length to some new values. An example of a perturbation JSON message is given in the section Perturbation below.
- **/query**: the TH issues a request to this endpoint to query the current internal state of the SUT.

These endpoints allow LL to interact with the system during runtime. We are considering making the full capability of the existing TCP socket API (that was implemented in Phase I, and extended in Phase II) available through the **/query** endpoint.

Note: Due to the unavailability of a Swift server stub code generator, the implementation of server part of the Swagger specification has been developed manually, and thus bugs related to inconsistency between specification and implementation are to be expected.

Client

As a **Client**, Proteus will post to the REST endpoints exposed by the Test Harness (TH). Below is a brief description of each of these endpoints. The details of the messages and their corresponding responses are specified in [THRestServer.yaml](#).

- **TH/ready**: when the SUT is ready for execution, Proteus will post a DAS_READY message containing the time the message is sent.
- **TH/error**: in case an error is encountered during the execution of the SUT, Proteus will post a DAS_ERROR message.
- **TH/done**: when the execution of the SUT successfully runs to completion, Proteus will post a DONE message similar to the one defined in Phase I.
- **TH/status**: SUT posts to this endpoints to report its current internal state. The JSON payload for this post is the same as that of the response to the SUT/query request by TH.

Initialization Parameters

An example of initialization parameters for CP1 can be found at JSON path `params.initialParms` [here](#).

Perturbation

Perturbation messages are JSON object of the same (OpenAPI 3.0) type as the `initialConditions` property of the initialization parameters object linked to above.

Intent Specification and Evaluation Metrics

In Phase I intents were hard-coded in the application. In contrast, in Phase II, intents will be specified in separate files, using the FAST Intent Specification Language. An example of such a specification for CP1 can be found [here](#). It includes both application knobs (`cdr`, `fdr`, `numberOfRanges`, `numberOfBeams`) as well as system configuration knobs (`utilizedCores`, `utilizedCoreFrequency`).

Appendix: Challenge Problem x264

Rice Proteus Challenge Problem 2: Video Encoding (x264)

Overview

The mission scenario is unchanged from Phase II. The context of the challenge problems is an embedded system, e.g. a UAV equipped with various sensors that is conducting a designated mission. The UAV must process the incoming data from the sensors and report its analysis back to base. The available resources may fluctuate, e.g., battery level, available uplink bandwidth, etc. The system will try to adapt to these changes by utilizing online control and machine learning supported by off-line machine learning in order to maintain the quality requirements posed by the base through our novel programming language FAST that includes application intent specifications in addition to conventional code. The aforementioned tasks are carried out by a dedicated on-board system (independent from the avionics control) running the corresponding intent-enriched application.

Compared to Phase II, the scenario is scaled up in two ways. First, additional complexity will be made possible in the `missionIntent` test parameter. Whereas in Phase II, encoded a constrained optimization problem with a single constraint of the form $m = c$, where m was a measure name and c a constant, in Phase III, constraints are generalized to arbitrary conjunctions of inequalities between measure names and constants. Second, the expressivity of the FAST library API will be expanded to support dynamic manipulation of the configuration space.

Together, the new functionalities overlap in expressivity with two test parameters that were demonstrated in Phase II (`sceneImportance`, and `missionLength` with its associated `enforceEnergyLimit` parameter), which will be removed in Phase III, to explore the ability to express corresponding constraints (on energy and quality, respectively) in the new, more general intent language and library API.

This challenge problem will focus on demonstrating that Proteus is able to meet multi-constraint intents by adapting combinations of system configuration knobs and application knobs, under dynamic changes to the configuration space, neither of which were demonstrated in Phase II.

Test Data

The test data to be used by the x264 application is the `highway.cif` video sequence from the [EvalVid reference collection](#), converted to the uncompressed YUV format.

Test Parameters

Three test parameters will be exposed for CP2. The first is `missionIntent`, which encodes a multi-constraint optimization problem, and describes the configurations that are available to the controller to solve this problem, in the form of "knob ranges" (lists of values that each knob can be set to). Both of these aspects of an intent can be modified at runtime.

The second and third test parameters (`availableCores` and `availableCoreFrequency`) constrain the permissible ranges of system configuration knobs, analogous to those used in Phase

II evaluation.

Name	Value	Description
missionIntent	IntentSpecification	Changes the intent that is to be met by Proteus. Value can be changed prior to or during test runtime. Instructions for generating interesting values for test are to be detailed.
availableCores	Integer in {2,3,4}	Change the number of CPU cores on which the application is running. Value can be changed prior to or during test runtime.
availableCoreFrequency	Integer in {600,1200}	Change the operating frequency of the CPU on which the application is running. Value can be changed prior to or during test runtime.

Table 32: Challenge Problem x264 Test Parameters

The syntax of the `missionIntent` test parameter is given by the [IntentSpecification](#) schema. It includes a set of *knob* declarations, which represent the different configurations that the system can enter to adapt to meet the mission intent, or to adapt to perturbations. For the purpose of Phase III evaluation, Proteus (and its emulation component) supports setting the knobs of both the application and the system to the values listed in the table below. The example in the section "Intent Specification and Evaluation Metrics" at the end of this document shows how the possible knob values are represented in an intent specification.

Name	Value	Description
me	Integer in {1,12,16}	Motion estimation range. Decreasing this knob should increase the performance measure, in exchange for a lower compression ratio.

Name	Value	Description
subme	Integer in {1,7,11}	Sub-pixel motion estimation complexity. The individual values of this knob correspond to different algorithms used during compression, and thus the tradeoffs exposed by this knob are complex . Generally, though, higher values of this parameter should correspond to a higher compression ratio and higher quality measure, in exchange for a lower performance measure.
reframes	Integer in {1,16}	Number of reference frames. Decreasing this knob should increase the performance measure, in exchange for a decrease in compression ratio.
utilizedCores	Integer in {2,3,4}	Number of CPU cores that the application should distribute its computation over. Increasing this knob should increase the performance measure, in exchange for a decrease in the powerConsumption/energy-Delta measures. See side note below.
utilizedCoreFrequency	Integer in {600,1200}	The CPU clock frequency that the system should be configured to. Increasing this knob should increase the performance measure, in exchange for a decrease in the powerConsumption/energyDelta measures. See side note below.

Table 33: Challenge Problem x264 Knobs

Side note about the utilizedCores and utilizedCoreFrequency knobs: The tradeoff exposed by these knobs is counter intuitive since, generally, higher clock frequencies require a higher voltage, and using more cores requires powering a bigger circuit, and thus should require more energy. However, due to the unthrottled nature of the Phase II CPs (in contrast to actual sensor inputs, that may be read at a fixed rate), the processing of iterations can happen at an arbitrarily high rate. This means that it can be optimal, from an energy consumption standpoint, to process all iterations at the highest possible frequency / most cores, allowing the system to complete the processing as soon as possible, since, at lower frequencies / less cores, energy is spent not only on powering the CPU, but also other system components (accounted for in the energy-derived measures). Consequently, whenever the intent specification mandates minimizing energy (or powerConsumption or energyDelta), Proteus will only use configurations that include low frequencies / few cores when it has no other choice (such as when the availableCoreFrequency test parameter specifies that a higher frequency is not available).

Test Procedure

In Phase I, the oracle compared Proteus to a clairvoyant non-adaptive system. It was constructed by running the system in ten Pareto-optimal configurations, and choosing the best one w.r.t. the intent. This served well to showcase the benefit of adaptation. However, it was not clear how much insight it gave into how good Proteus is at adaptation.

To improve on this, we have developed an alternative oracle, whose behavior illustrated in Figure 33 on page 98.

In this plot, the vertical axis represents the constraint value (in this case, the window performance, that is, the number of inputs processed per second). The green line represents the goal stated in the intent specification. The orange line represents the window average of the observed performance of Proteus, executing with adaptation enabled. The pink line represents the performance of the Oracle, which is constructed from logs gathered during a set of executions, each corresponding to a fixed configuration. Each of the grayscale lines in the plot corresponds to one such execution. Note that, although the system is running in a fixed configuration for each of these executions, the observed measure (performance in this case) values fluctuate. This is due to several factors, including variability in system resources (e.g. operating system process scheduling) and the possibly input-dependent nature of the application (a frame of a video may take a different amount of time/resources to process).

The oracle is defined in terms of several parameters:

- `GLOBAL_CONSTRAINT_ERROR_THRESHOLD` (GCET) A threshold on the cumulative relative absolute error (versus the constraint goal), below which an execution is considered to have met the constraint.
- `GLOBAL_OBJECTIVE_FUNCTION_ERROR_THRESHOLD` (GOFET) A threshold on the objective function error (versus the oracle), below which a Proteus execution is considered to be sufficiently close to the optimum.
- `LOCAL_ERROR_THRESHOLD` (LET) A threshold on the relative window error (versus the constraint goal), below which a configuration is considered to meet the *equality* constraint (at a given input).
- `WINDOW_SIZE` (WS) The number of inputs over which window averages are computed.
- `STABILIZATION_PERIOD` (SP) The number of inputs after a perturbation (including the start of the execution) that are excluded from the verdict computation. This is equal to 200 for all CPs.

The values of the first three of the above parameters are specific to this challenge problem:

Parameter	CP2
<code>GLOBAL_CONSTRAINT_ERROR_THRESHOLD</code>	0.2
<code>GLOBAL_OBJECTIVE_FUNCTION_ERROR_THRESHOLD</code>	0.30
<code>LOCAL_ERROR_THRESHOLD</code>	0.1
<code>WINDOW_SIZE</code>	40

Table 34: Challenge Problem x264 Verdict Expression Parameters

The oracle is then used to define a verdict expression.
In the sections below:

- MARE stands for mean absolute relative error.
- FCE stands for fixed-configuration execution.

Oracle

The oracle attempts to meet the *equality* constraint *at each input processed during the execution*. It is meant to resemble Proteus by adapting to changes in the `missionIntent`, unlike the Phase I oracle, which represented an idealized non-adaptive system. The Phase III oracle is constructed as follows. For each input, the oracle has the behavior (the values and statistics) of one of the FCEs at that input. If, for this input, none of the FCEs meet the constraint (their MARE exceeds `LOCAL_ERROR_THRESHOLD`), then select the behavior of the FCE with the smallest MARE. Otherwise, select the one of those FCEs that meet the constraint with the best objective function, that is, highest or lowest, depending on the optimization type (min or max) given in the intent specification at the corresponding input.

Verdict Expression

The verdict expression is defined based on the oracle as illustrated in Figure 34 on page 100, where:

- e_p and e_o stand for the combined mean relative errors of the Proteus and oracle executions, respectively.
- The relative objective function oracle advantage (ROFOA), which is a measure of how much better the oracle's objective function is compared to that of Proteus (set to 0 when Proteus surpasses the oracle), is defined as follows. Let f_p and f_o be the normalized objective functions of the Proteus and oracle executions, respectively (if the optimization type is min, then the original objective function is negated to give the normalized objective function; if it is max, the normalized objective function is equal to the original objective function). Further let $f_{o,min}$ and $f_{o,max}$ be the minimal and maximal normalized objective functions counted across all oracle executions. Then, define ROFOA as 0 if $f_o \leq f_p$ and $(f_o - f_p) / (f_{o,max} - f_{o,min})$ otherwise.

Interface to the Test Harness (API)

In Phase III, Proteus will be communicating with the Test Harness over RESTful APIs in two ways:

- a **Server** (part of the Proteus runtime) that the Test Harness can call to query and perturb the state of a Challenge Problem application;
 - a **Client** that is used to call REST endpoints exposed by the Test Harness.
1. **TH**: Starts the SUT (a particular CP application).
 2. **SUT**: Starts the Proteus REST server.
 3. **SUT**: Posts to TH/ready, and receives initialization message in the response. An example of the JSON initialization message is given in the section Initial Parameters below.

4. **SUT**: Initializes the Proteus runtime with respect to the response from TH/ready.
5. **SUT**: Posts to TH/initialized.
6. Loop until end of input stream (**either of these could happen at any time**):
 - TH drives the test one "step" forward (**by posting to either of these**):
 - **TH**: Posts to SUT/process with a message containing the number of inputs to process.
 - **TH**: Posts to SUT/perturb with a message containing the test parameters. The **SUT** responds synchronously with a message containing the status after the perturbation has been applied.
 - **SUT**: Posts to TH/status after the processing of each input.
 - **TH**: Gets from SUT/alive to obtain SUT liveness information.
 - **TH**: Gets from SUT/query to obtain SUT status.
7. **SUT**: Posts to TH/done to signal that all the inputs in the input stream have been processed.

Server

As a **Server**, Proteus will expose a RESTful API with four endpoints. We use the formal specification standard [Swagger](#) to document the API. The Proteus REST API ([RestServer.yaml](#)) is available as part of the FAST source code. The main constituents of the specification are the paths, which specify the REST endpoints of the API, and the definitions, which specify the types used for input and output parameters.

The paths are as follows:

- **/alive**: the TH issues a request to this endpoint to query the liveness of the SUT.
- **/process**: the TH issues a request to this endpoint to have the SUT process a number of inputs as specified in the request JSON payload.
- **/perturb**: the TH issues a request to this endpoint with a JSON message containing appropriate entries defining the perturbations to which the SUT must respond. An example perturbation may set the performance goal to some new value. An example of a perturbation JSON message is given in the section Perturbation below.
- **/query**: the TH issues a request to this endpoint to query the current internal state of the SUT.

These endpoints allow SwRI to interact with the system during runtime. We are considering making the full capability of the existing TCP socket API (that was implemented in Phase I, and extended in Phase II) available through the **/query** endpoint.

Note: Due to the unavailability of a Swift server stub code generator, the implementation of server part of the Swagger specification has been developed manually, and thus bugs related to inconsistency between specification and implementation are to be expected.

Client

As a **Client**, Proteus will post to the REST endpoints exposed by the Test Harness (TH). Below is a brief description of each of these endpoints. The details of the messages and their corresponding responses are specified in [THRestServer.yaml](#).

- **TH/ready:** when the SUT is ready for execution, Proteus will post a `DAS_READY` message containing the time the message is sent.
- **TH/error:** in case an error is encountered during the execution of the SUT, Proteus will post a `DAS_ERROR` message.
- **TH/done:** when the execution of the SUT successfully runs to completion, Proteus will post a `DONE` message similar to the one defined in Phase I.
- **TH/status:** SUT posts to this endpoints to report its current internal state. The JSON payload for this post is the same as that of the response to the SUT/query request by TH.

Initialization Parameters

An example of initialization parameters for CP2 can be found at JSON path `params.initialParms` [here](#).

Perturbation

Perturbation messages are JSON object of the same (OpenAPI 3.0) type as the `initialConditions` property of the initialization parameters object linked to above.

Intent Specification and Evaluation Metrics

As in Phase II, intents will be specified in separate files, using the FAST Intent Specification Language. For Phase III, the syntax for the intent file is slightly modified for improved readability. An example of such a specification for CP2 can be found [here](#). It includes constraints on both performance and quality, while optimizing the energy used to process a frame application.

Appendix: Challenge Problem CaPSuLe

Rice Proteus Challenge Problem 3: Image Matching (CaP-SuLe)

Overview

The mission scenario is the same as that described in the overview section of Appendix: Challenge Problem x264 on page [103](#).

The image matching problem is based on research conducted by members of the Proteus team. CaPSuLe distinguishes itself as a challenge problem compared to the CP2 (x264) application by: making largely input-independent computations, leading to more predictable behavior, which makes it easier to isolate the behavior the Proteus runtime; relying on a compressed database of images that must be available to the algorithm. The accuracy (matching rate) of the algorithm can be improved by increasing the size of the database, meaning that memory usage is a bottleneck in this application, unlike x264. While the x264 algorithm comes with a readily-accessible and self-contained quality measure (peak signal-to-noise ratio), the accuracy of CaPSuLe is defined as the maximum distance between rankings of matches reported by the application, a measure of the kurtosis of the corresponding distribution of matches.

The scenario enables flexibility along two dimensions. First, in the `missionIntent` test parameter, constraints are generalized to arbitrary conjunctions of inequalities between measure names and constants. Second, the expressivity of the FAST library API supports dynamic manipulation of the configuration space.

This challenge problem focuses on demonstrating that Proteus is able to meet multi-constraint intents by adapting combinations of system configuration knobs and application knobs, under dynamic changes to the configuration space, neither of which were demonstrated in Phase II.

Unlike CP2, this challenge problem may require adding new knobs and/or measures to the base application (CaPSuLe) to effectively demonstrate dynamic changes to the configuration space. The degree to which these knobs will expose useful tradeoffs in the existing and/or new measures will determine whether this functionality will be part of the challenge problem.

Test Data

The test data to be used for CP3 is a synthesized video sequence from a drone, along with a database of aerial photographs. The image matching performed by CaPSuLe thus corresponds to solving a localization problem.

Test Parameters

Three test parameters will be exposed for CP3. The first is `missionIntent`, which encodes a multi-constraint optimization problem, and describes the configurations that are available to the controller to solve this problem, in the form of "knob ranges" (lists of values that each knob can be set to). Both of these aspects of an intent can be modified at runtime.

The second and third test parameters (`availableCores` and `availableCoreFrequency`) con-

strain the permissible ranges of system configuration knobs, analogous to those used in Phase II evaluation.

Name	Value	Description
missionIntent	IntentSpecification	Changes the intent that is to be met by Proteus. Value can be changed prior to or during test runtime. Instructions for generating interesting values for test are to be detailed.
availableCores	Integer in {2,4}	Change the number of CPU cores on which the application is running. Value can be changed prior to or during test runtime.
availableCoreFrequency	Integer in {600,1200}	Change the operating frequency of the CPU on which the application is running. Value can be changed prior to or during test runtime.

Table 35: Challenge Problem CaPSuLe Test Parameters

The syntax of the `missionIntent` test parameter is given by the [IntentSpecification](#) schema. It includes a set of *knob* declarations, which represent the different configurations that the system can enter to adapt to meet the mission intent, or to adapt to perturbations. For the purpose of Phase III evaluation, Proteus (and its emulation component) supports setting the knobs of both the application and the system to the values listed in the table below. The example in the section "Intent Specification and Evaluation Metrics" at the end of this document shows how the possible knob values are represented in an intent specification.

Name	Value	Description
numberOfFeatures	Integer in {16,512,1024}	Number of features used to match images. Increasing this knob should increase the quality measure, in exchange for a decrease in the performance measure.

Name	Value	Description
numberOfHashes	Integer in {6,22}	Number of hashes used to build the image matching database. Increasing this knob should increase the performance measure, in exchange for a decrease in the quality measure.
numberOfHashTables	Integer in {8,24}	Number of hash tables used to build the image matching database. Increasing this knob should increase the quality measure, in exchange for a decrease in the performance measure.
utilizedCores	Integer in {2,4}	Number of CPU cores that the application should distribute its computation over. Increasing this knob should increase the performance measure, in exchange for a decrease in the powerConsumption/energyDelta measures. See side note below.
utilizedCoreFrequency	Integer in {600,1200}	The CPU clock frequency that the system should be configured to. Increasing this knob should increase the performance measure, in exchange for a decrease in the powerConsumption/energyDelta measures. See side note below.

Table 36: Challenge Problem CaPSuLe Knobs

Side note about the utilizedCores and utilizedCoreFrequency knobs: The tradeoff exposed by these knobs is counter intuitive since, generally, higher clock frequencies require a higher voltage, and using more cores requires powering a bigger circuit, and thus should require more energy. However, due to the unthrottled nature of the Phase II CPs (in contrast to actual sensor inputs, that may be read at a fixed rate), the processing of iterations can happen at an arbitrarily high rate. This means that it can be optimal, from an energy consumption standpoint, to process all iterations at the highest possible frequency / most cores, allowing the system to complete the processing as soon as possible, since, at lower frequencies / less cores, energy is spent not only on powering the CPU, but also other system components (accounted for in the energy-derived measures). Consequently, whenever the intent specification mandates minimizing energy (or powerConsumption or energyDelta), Proteus will only use configurations that include low frequencies / few cores when it has no other choice (such as when the availableCoreFrequency test parameter specifies that a higher frequency is not available).

Test Procedure

The test procedure for CP3 is the same as that described in the [announcement document for CP2](#) (x264), modulo the following specific values that occur in the verdict expression:

Parameter	CP3
GLOBAL_CONSTRAINT_ERROR_THRESHOLD	0.1
GLOBAL_OBJECTIVE_FUNCTION_ERROR_THRESHOLD	0.15
LOCAL_ERROR_THRESHOLD	0.05
WINDOW_SIZE	20

Table 37: Challenge Problem CaPSuLe Verdict Expression Parameters

Interface to the Test Harness (API)

The Test Harness API for CP3 is the same as that described in the [announcement document for CP2](#) (x264).

Initialization Parameters

An example of initialization parameters for CP3 can be found at JSON path `params.initialParms` [here](#).

Perturbation

Perturbation messages are JSON object of the same (OpenAPI 3.0) type as the `initialConditions` property of the initialization parameters object linked to above.

Intent Specification

Intents are specified in separate files, using the FAST Intent Specification Language. In Phase III, the syntax for the intent file was slightly modified for improved readability. An example of such a specification for CP3 can be found [here](#). It includes constraints on both performance and quality, while optimizing the energy used to process a frame application.

Appendix: Challenge Problem flightTestScenario7

Rice Proteus Phase III Challenge Problem 1: Flight Test Scenario 7 (flightTestScenario7)

Overview

The context of this challenge problem is a flight test scenario involving three aircraft (test articles) TA1, TA2, and TA3 that share the same radio frequency during a designated mission on a test range. Unforeseen conditions may arise that prevent a particular test article from using all of its allocated bandwidth. For example, during the mission, TA3 may fly behind a mountain range that prevents it from using its allocated bandwidth to complete its own test objectives. The goal in this challenge problem is to reallocate the available bandwidth to TA1 and TA2, so that it may take advantage of it to conduct additional tests, until TA1 needs its allocated bandwidth again.

Proteus will adapt to these changes by utilizing online control and machine learning supported by off-line machine learning in order to maintain operational requirements posed by the base through our novel programming language FAST that includes application intent specifications in addition to conventional code.

This challenge problem will focus on demonstrating that Proteus is able to meet the mission intent by adapting combinations of application knobs. In a Mission Control instance specified by SwRI, Proteus is assumed to be executing in the System Manager (SM) component and provides the appropriate dynamic scheduling to the Link Manager (LM) by writing configuration data to the SwRI OrientDB database.

Test Data

The part of this challenge problem that is controlled by Proteus does not take any input data.

Test Parameters

Test parameters will be exposed for CP1 in the form of `missionIntent`, which encodes a constrained optimization problem, and describes the configurations that are available to the controller to solve this problem, in the form of "knob ranges" (lists of values that each knob can be set to). Both of these aspects of an intent can be modified at runtime.

Name	Value	Description
missionIntent	IntentSpecification	Changes the intent that is to be met by Proteus. Value can be changed prior to or during test runtime. Instructions for generating interesting values for test are to be detailed.

Table 38: Challenge Problem flightTestScenario7 Test Parameters

The syntax of the `missionIntent` test parameter is given by the [IntentSpecification](#) schema. It includes a set of *knob* declarations, which represent the different configurations that the system can enter to adapt to meet the mission intent, or to adapt to perturbations. For the purpose of Phase III evaluation, Proteus (and its emulation component) supports setting the knobs of both the application and the system to the values listed in the table below. The example in the section "Intent Specification and Evaluation Metrics" at the end of this document shows how the possible knob values are represented in an intent specification.

Name	Value	Description
ta1	Integer in {0, 600, 1200, 2000, 4000}	Transmission bandwidth for TA1.
ta2	Integer in {0, 600, 1200, 2000, 4000}	Transmission bandwidth for TA2.
ta3	Integer in {0, 1200, 2000, 4000}	Transmission bandwidth for TA3.
ground1	Integer in {0, 200, 400}	Ground transmission bandwidth associated with TA1.
ground2	Integer in {0, 200, 400}	Ground transmission bandwidth associated with TA2.
ground3	Integer in {0, 400, 600}	Ground transmission bandwidth associated with TA3.

Table 39: Challenge Problem flightTestScenario7 Knobs

The knob ranges are subject to a static constraint, that is used to enforce relationships between the knobs. In the case of CP1, the constraints enforce a relationship between the bandwidth allocated to a particular TA's radio, and its ground station radio. The CP1 static constraints are as follows:

```
[ta1, ground1] from [ [0, 0], [600, 200], [1200, 200], [2000, 200], [4000, 400] ]
and
[ta2, ground2] from [ [0, 0], [600, 200], [1200, 200], [2000, 200], [4000, 400] ]
and
[ta3, ground3] from [ [0, 0], [1200, 400], [2000, 400], [4000, 600] ]
```

```

and
ta1 + ground1 + ta2 + ground2 + ta3 + ground3 <= 10000
and
ta1 + ground1 + ta2 + ground2 + ta3 + ground3 > 0

```

where e from l is true when the list l contains the element e .

Test Procedure

The test procedure for CP1 is the same as that described in the [announcement document for CP2](#) (x264), modulo the following specific values that occur in the verdict expression:

Parameter	CP1
GLOBAL_CONSTRAINT_ERROR_THRESHOLD	0.2
GLOBAL_OBJECTIVE_FUNCTION_ERROR_THRESHOLD	0.25
LOCAL_ERROR_THRESHOLD	0.1
WINDOW_SIZE	20

Table 40: Challenge Problem flightTestScenario7 Verdict Expression Parameters

Interface to the Test Harness (API)

The evaluation of CP1 will be facilitated by automating tests using a custom test harness. This application consists of two components that execute alongside CP1 and the [RadioQueue simulator](#), to orchestrate the scenario described at the beginning of this document:

1. A [TA simulator](#). This component simulates the behavior of a TA's radio, by observing the bandwidth allocated to the radio in OrientDB, and gradually adjusting the utilized bandwidth until it matches the allocated bandwidth.
2. A [scenario runner](#). This component reads a scenario specification file, consisting of a sequence of permutations, and applies the permutations at given points in time. Each permutation specifies which of the TAs are currently online, and is applied by updating the corresponding node in the database.

To execute a scenario, an additional [script](#) is used to bring up OrientDB, CP1, the RadioQueue simulator, the TA simulator and the scenario runner.

Intent Specification and Evaluation Metrics

Intents will be specified in separate files, using the FAST Intent Specification Language. An example of such a specification for CP1 can be found [here](#).

The application will expose the relevant parts of its state through the Test Harness API to let SwRI measure intent preservation.

Bibliography

- [1] Adam Duracz and Ferenc A. Bartha and Dung X. Nguyen and Ryuichi Sai and Yao-Hsiang Yang. Source code for FAST. <https://github.mit.edu/proteus/FAST>.
- [2] Adam Duracz and Ferenc A. Bartha and Dung X. Nguyen and Ryuichi Sai and Yao-Hsiang Yang and Jason Miller. Source code for all challenge problems. <https://github.mit.edu/brass/rice-proteus>.
- [3] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):7, 2014.
- [4] Apple. Open-source swift github repository. <https://github.com/apple/swift/>.
- [5] Apple. The swift programming language. <https://swift.org/>.
- [6] Apple. Swift standard library linux implementation status. <https://github.com/apple/swift-corelibs-foundation/blob/master/Docs/Status.md>.
- [7] ARM. *Cortex-A9 MPCore Programmer Advice Notice*. ARM, 2011.
- [8] ARM. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM, 2016.
- [9] Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton, NJ, USA, 2008.
- [10] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.
- [11] K. Bergman. Nanophotonic Interconnection Networks in Multicore Embedded Computing. In *2009 IEEE/LEOS Winter Topicals Meeting Series*, pages 6–7, Jan 2009.
- [12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.
- [13] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC ’14, pages 7:1–7:6, New York, NY, USA, 2014. ACM.
- [14] Léon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT’2010)*, pages 177–186, Heidelberg, 2010. Physica-Verlag HD.

- [15] Ashesh Chattopadhyay, Pedram Hassanzadeh, Krishna Palem, and Devika Subramanian. Data-driven prediction of a multi-scale lorenz 96 chaotic system using a hierarchy of deep learning methods: Reservoir computing, ann, and rnn-lstm. *arXiv preprint arXiv:1906.08829*, 2019.
- [16] Ian G. Cumming and Frank H. Wong. *Digital processing of synthetic aperture radar data : algorithms and implementation*. Artech House remote sensing library. Artech House, Boston, 2005.
- [17] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Fred Chong. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvm. In *MICRO*, 2017.
- [18] Yi Ding, Nikita Mishra, and Henry Hoffmann. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 39–52. ACM, 2019.
- [19] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.
- [20] Matthew Faulkner, Michael Olson, Rishi Chandy, Jonathan Krause, K Mani Chandy, and Andreas Krause. The next big one: Detecting earthquakes and other rare events from community-based sensors. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 13–24. IEEE, 2011.
- [21] Anne-Kathrin Hess and Anders Rantzer. Distributed Kalman Filter Algorithms for Self-localization of Mobile Devices. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, pages 191–200, New York, NY, USA, 2010. ACM.
- [22] Ralf Hildebrandt and Andreas Zeller. Simplifying Failure-Inducing Input. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 135–145, New York, NY, USA, 2000. ACM.
- [23] Henry Hoffmann. Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 223–232. IEEE, 2014.
- [24] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann. POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86, April 2015.
- [25] Connor Imes and Henry Hoffmann. Bard: A unified framework for managing soft timing and power constraints. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 31–38. IEEE, 2016.

- [26] Connor Imes and Henry Hoffmann. Bard: A unified framework for managing soft timing and power constraints. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016.
- [27] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. Poet: A portable approach to minimizing energy under soft real-time constraints. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 75–86. IEEE, 2015.
- [28] K. Mani Chandy. Assemble software to sense and respond to critical events. <https://www.assemblesoftware.com>.
- [29] K. Mani Chandy and Adam Duracz and Ryuichi Sai. Source code for proof-of-concept assurance monitor. <https://github.mit.edu/proteus/assurance-monitor-poc>.
- [30] Rudolph Emil Kalman. A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME—Journal of Basic Engineering*, 82(Series D):35–45, 1960.
- [31] David H. K. Kim, Connor Imes, and Henry Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *CPSNA*, 2015.
- [32] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.
- [33] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. Automated control of multiple software goals using multiple actuators. In *ES-EC/FSE*, 2017.
- [34] Jason Miller and Connor Imes. Energymon github repository, zcu102 implementation. <https://github.com/energymon/energymon/tree/master/zcu102>.
- [35] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. CALOREE: learning control for predictable latency and low energy. In *ASPLOS*, 2018.
- [36] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. Controlling ai engines in dynamic environments. In *SysML*, 2018.
- [37] Yongshik Moon, Soonhyun Noh, Daedong Park, Chen Luo, Anshumali Shrivastava, Seongsoo Hong, and Krishna Palem. Capsule: A camera-based positioning system using learning. In *System-on-Chip Conference (SOCC), 2016 29th IEEE International*, pages 235–240. IEEE, 2016.
- [38] Eugene Morelli. System identification programs for aircraft (sidpac). *AIAA Atmospheric Flight Mechanics Conference and Exhibit*, 2002.
- [39] Kaushik Nagarajan, Nicholas Gans, and Roozbeh Jafari. Modeling Human Gait Using a Kalman Filter to Measure Walking Distance. In *Proceedings of the 2nd Conference on Wireless Health, WH ’11*, pages 34:1–34:2, New York, NY, USA, 2011. ACM.
- [40] R. E. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, 2003.

- [41] Raghavendra Pradyumna Pothukuchi, Amin Ansari, Petros Voulgaris, and Josep Torrellas. Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 658–670. IEEE, 2016.
- [42] John R Quinlan et al. Learning with continuous classes. In *5th Australian joint conference on artificial intelligence*, volume 92, pages 343–348. World Scientific, 1992.
- [43] Prasad Raghavendra. Optimal algorithms and inapproximability results for every csp? In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 245–254. ACM, 2008.
- [44] I. E. Richardson. *The H.264 Advanced Video Compression Standard*. Signal Processing. Wiley, New York, 2010.
- [45] Martin C. Rinard. Using Early Phase Termination To Eliminate Load Imbalances At Barrier Synchronization Points. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07*, pages 369–386, New York, NY, USA, 2007. ACM.
- [46] The risc-v instruction set. <https://riscv.org/>.
- [47] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning Assistant for Floating-Point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13*, pages 27:1–27:12, New York, NY, USA, 2013. ACM.
- [48] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *ACM SIGPLAN Notices*, volume 46, pages 175–186. ACM, 2011.
- [49] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Optimization of Floating-Point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 53–64, New York, NY, USA, 2014. ACM.
- [50] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer’s model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.
- [51] Richard Smith, editor. *Working Draft, Standard for Programming Language C++*. <http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4527.pdf>, May 2015.
- [52] Éfren L. Souza, Eduardo F. Nakamura, and Richard W. Pazzi. Target Tracking for Sensor Networks: A Survey. *ACM Comput. Surv.*, 49(2):30:1–30:31, June 2016.
- [53] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. Proactive Control of Approximate Programs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16*, pages 607–621, New York, NY, USA, 2016. ACM.

- [54] Xin Sui, Andrew Lenharth, Donald S Fussell, and Keshav Pingali. Proactive control of approximate programs. *ACM SIGOPS Operating Systems Review*, 50(2):607–621, 2016.
- [55] Swarnendu Biswas, Yan Pei, Donald S. Fussell, and Keshav Pingali. Capri: A Control System for Approximate Systems. *ArXiv e-prints*, June 2017.
- [56] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.
- [57] Michel Tokic. Adaptive ϵ -greedy exploration in reinforcement learning based on value differences. In *Annual Conference on Artificial Intelligence*, pages 203–210. Springer, 2010.
- [58] Umberto Raimondi. Swift on arm github repository. <https://github.com/uraimo/buildSwiftOnARM>.
- [59] L. G. Valiant. A Theory of the Learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.
- [60] Shu Wang, Chi Li, William Sentosa, Henry Hoffmann, Shan Lu, and Achmad Imam Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. In *ASPLOS*, 2018.
- [61] Greg Welch and Gary Bishop. An Introduction to the Kalman Filter. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.
- [62] Joyce Jiyoung Whang, Xin Sui, and Inderjit S. Dhillon. Scalable and Memory-Efficient Clustering of Large-Scale Social Networks. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining, ICDM '12*, pages 705–714, Washington, DC, USA, 2012. IEEE Computer Society.
- [63] Yan Pei, Swarnendu Biswas, Donald S. Fussell, and Keshav Pingali. An Elementary Introduction to Kalman Filtering. *ArXiv e-prints*, October 2017.
- [64] Yan Pei, Swarnendu Biswas, Donald S. Fussell, and Keshav Pingali. An Elementary Introduction to Kalman Filtering. *Communications of the ACM*, 2019.
- [65] Yao-Hsiang Yang, Adam Duracz, Ferenc A Bartha, Ryuichi Sai, Ahsan Pervaiz, Saeid Barati, Dung Nguyen, Robert Cartwright, Henry Hoffmann, and Krishna V Palem. Language support for adaptation: Intent-driven programming in FAST. *arXiv preprint arXiv:1907.08695*, 2019.
- [66] Yongshik Moon and Soonhyun Noh and Daedong Park and Chen Luo and Anshumali Shrivastava and Seongsoo Hong and Krishna Palem and Adam Duracz and Ryuchi Sai. C++ source code for capsule. <https://github.mit.edu/proteus/capsule>.
- [67] Sizhuo Zhang. *Constructing and evaluating weak memory models*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 2019.
- [68] Sizhuo Zhang, Arvind, and Muralidaran Vijayaraghavan. Taming weak memory models. *arXiv preprint arXiv:1606.05416*, 2016.

- [69] Sizhuo Zhang, Muralidaran Vijayaraghavan, and Arvind. Weak memory models: Balancing definitional simplicity and implementation flexibility. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 288–302, Sept 2017.
- [70] Sizhuo Zhang, Muralidaran Vijayaraghavan, Andrew Wright, Mehdi Alipour, and Arvind. Constructing a weak memory model. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 124–137, June 2018.
- [71] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind. Composable building blocks to open up processor design. In *2018 51st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2018.

List of Symbols, Abbreviations and Acronyms

Azimuth (width / beam)
Function composition
Energy
The natural numbers
Range (depth)
Pole of a controller
Time
64-bit ARM architecture
ARM calling convention
Arithmetic Logic Unit
Artificial Neural Network
Application Programmer Interface
Advanced RISC Machine
Architectural Support for Programming Languages and Operating Systems
ARM Trusted Firmware
Heterogeneous processing architecture by ARM
Building Resource Adaptive Software Systems
Energy-efficient camera-based Positioning System
Commercial Off the Shelf
Challenge Problem
Central Processing Unit
Constraint Satisfaction Problem
Comma Separated Value
DataBase
Entity Relationship
Not an acronym
Fixed-Configuration Execution
Floating Point
Field-Programmable Gate Array
First-Stage Boot Loader

GAM	General Atomic Memory Model
GCET	Global Constraint Error Threshold
GEM5	Platform for computer-system architecture research
GHz	GigaHertz
GitHub	Source code repository
GOFET	Global Objective Function Error Threshold
GPGPU	General Purpose Graphics Processing Unit
GPS	Global Positioning System
H.264	Video encoding standard
HBASE	Not an acronym
I ² E	Instantaneous Instruction Execution
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
iOS	Mobile operating system created and developed by Apple Inc.
IoTPy	Framework for event-based systems
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
ITU	Telecommunication Standardization Sector
JSON	JavaScript Object Notation
Keras	Machine learning library
LET	Local Error Threshold
LL	Lincoln Laboratory
LM	Link Manager
LPF	Low-pass Filter
M5	Regression model
MARE	Mean Absolute Relative Error
MCRCE	Mean Combined Relative Constraint Error
MetaFAST	Not an acronym
MIT	Massachusetts Institute of Technology
ML	Machine Learning
MLP	Multi-Layer Perceptron
MPEG	Moving Pictures Experts Group
MPSoC	MultiProcessor System-on-a-Chip

MSHR	Miss Status Holding Register
NOP	No-operation assembly language instruction
ODROID	Single-board computers by Hardkernel Co., Ltd.
OOO	Out-Of-Order
OOTA	Out-Of-Thin-Air
OrientDB	Graph database
PAC	Probably Approximately Correct
PARSEC	Princeton Application Repository for Shared-Memory Computers
PEMU	Proteus platform emulator
PHT	Performance Hash Table
POET	Performance with Optimal Energy Toolkit
POST	Request method supported by HTTP
POWER	Instruction set architecture by IBM
PPO	Preserved Program Ordering
Proteus	Not an acronym
RAM	Random Access Memory
RDFT	Recursive Discrete Fourier Transform
REST	REpresentational State Transfer
RESTful	Web services that conform to the REST architectural style
RISC	Reduced Instruction Set Computer
RISC-V	An open-source hardware instruction set architecture
RLS	Recursive Least Squares
RMO	Relaxed Memory Ordering
ROB	ReOrder Buffer
ROFOA	Relative Object Function Oracle Advantage
ROM	Read-Only Memory
SAR	Synthetic Aperture Radar
SC	Sequential Consistency
SD	Secure Digital
SDP	SemiDefinite Programming
SDR	Software-Defined Radio
SIDPAC	System Identification Programs for Aircraft
SM	System Manager

SP	Stability Period
SPEC	Standard Performance Evaluation Corporation
SQ	Store Queue
SQL	Structured Query Language
STI	Settling Time Interval
SUT	System Under Test
Swift	Not an acronym
SwRI	Southwest Research Institute
SysML	Systems Modeling Language
TA*	Task Area *
TCP	Transmission Control Protocol
Tensorflow	Machine learning library by Google
TH	Test Harness
Theano	Machine learning library
TSO	Total Store Ordering
UAV	Unmanned Aerial Vehicle
URI	Uniform Resource Identifier
YUV	Not an acronym
WMM	Weak Memory Model
WS	Window Size
X264	H.264 Video Encoder
x86	Instruction set architecture by Intel
ZCU102	Embedded system evaluation board by Xilinx