Multiobjective System-Level Optimization for Networked Embedded Systems

Honglei Li
Department of Electrical and Computer Engineering
University of Maryland College Park
College Park, USA
lihonglei001@gmail.com

Supervisor in UMD: Prof. Shuvra S. Bhattacharyya, Ph.D.

Supervisor in FHS: DI (FH) DI Simon Kranzer

August 2018

CONTENTS

I Acknowledgments 4

II Introduction 5

III Related Work 6

III-A Network Simulation 6

III-B Computer Vision Application 7

IV Background 7

IV-A Dataflow Design Tools 7

IV-A1 DICE 7

IV-A2 LIDE 8

IV-B Application Design Tools 8

IV-B1 Native Development Kit and Android Debug Bridge 9

IV-B2 OpenCV4Android SDK 9

IV-C Network Simulator 9

V Case Study 10

V-A Simple Factory 10

V-A1 Simple Factory modeling 10

V-A2 Factory Simulator Package 12

V-A3 Coordination Module 14

V-A4 τLIDE 15

V-A5 Communication Interface Actors 17

V-A6 Factory Simulator Package Usage 18

V-B Hyperspectral Image Processing 23

VI Experiments 25

VI-A Factory Cosimulator 25

VI-A1 Simulation Parameters 25
VI-A2 Experiments with Different Protocols 27
VI-A3 Scalability Experiments 27
VI-B Hyperspectral Image Processing 31

VII Conclusion 34

References 34

I. ACKNOWLEDGMENTS

First and foremost, I thank the Marshall Plan Foundation for offering the research exchange scholarship and the instruction for the application to the scholarship.

I would like to thank Professor Shuvra Bhattacharyya for the suggestion of the great opportunity and his guidance throughout my research process.

I would like to thank FH DI. Simon Kranzer and Prof. DI Dr. Gerhard Jochtl in FH Salzburg for their support during my staying in FHS and their help in my life in Salzburg.

I would like to thank Dorian, Max, Gregor, and Georg for helping me and show me around the FH Salzburg. I really have happy experience with their help in the office and campus.

II. Introduction

In this research, we develop new methods for modeling, simulating, and analyzing networked embedded systems to aid in system-level optimization. The new system are based on dataflow modeling and wireless communication network simulator. The simulation results show the varying of communication efficiency with different network structure and parameters, which brings efficient ways for integrating dataflow models of embedded processing with state-of-the-art techniques for modeling and simulation of computer networks. Hyperspectral image processing applications is developed as future possible applications for the networked embedded system. In the application, a dynamic data driven hyperspectral video processing system is implemented on an Android device. The networked embedded system could support the communication between the Android devices with high efficiency and low latency.

The development in manufacturing system brings the requirement for stable wireless network as a supporting communication system. There exist many consideration aspects in the design of the wireless communication system including time latency, reliability, energy consumption and scalability. This special environment in the factory creates challenges to the design of the factory communication systems. The implementation of the wireless network has flexible possibility of communication standards such as IEEE80211a/ac/b/g/n. The design of the factory communication systems involves many parameters including the modulation type, coding rate, data rate and rate control manager. This research explores the design space and optimize the parameters based on the metrics evaluation.

To facilitate such design space exploration, we develop a factory simulator software package for simulating networked factory systems. This package is built on top of our previously developed Lightweight Dataflow Environment (LIDE) tool and the network simulator NS3. We use NS3 to simulate the wireless network through which these communication operations are carried out. To evaluate the performance a testbed is constructed that enables accurate and efficient experimentation with protocols for

industrial wireless networks. Perturbations to latency and reliability, which arise due to different provisions for real-time operation in wireless communication protocols, is observed.

In addition to the importance of the communication in manufacturing system, networked embedded systems encompass many important applications areas including applications in healthcare, security, defense, and industrial automation. Two application areas are to considered in this work are factory wireless networks and distributed computer vision systems. The application of hyperspectral image processing in Android system is also discussed in this report.

III. Related Work

A. Network Simulation

Recently, there are reported simulation method for the networked factory automation systems. For example, Liu et al. present a simulation framework that integrates process control system modeling and wireless network modeling [7]. This work focus on a discrete event simulator for the factory control modeling and their influence on the wireless communication. Won et al. present a tool that integrates the NS-2 (Network Simulator 2) and the dataflow tool for embedded signal processing [13].

The work in this report focus on dataflow-based modeling of factory process flows, and integrating the discrete events of factory control process for communication network simulation. With the developed cosimulation system, the communication efficiency of the embedded networked system could be simulated regarding to different network modeling and parameters. A factory simulator package is developed which support the interface between dataflow modeled discrete factory event simulator and the discrete communication simulator. The discrete communication simulator we used is Network Simulator 3 (NS-3).

B. Computer Vision Application

Hyperspectral sensor technology plays increasingly important roles in a variety of applications for monitoring and classifying in remote ground sensing, such as land cover classification. Recently the advances in sensor technology provide us with abundant datasets, which makes the research in hyperspectral more valuable and reliable. [3]. Hyperspectral imaging offers increased spectral diversity compared to traditional red-green-blue (RGB) channels. The total number of bands involved could be up to hundreds, thousands or even more bands.

Recently, Benezeth et al. have reported an background subtraction application of hyperspectral video and introduced a publicly available dataset [1]. Uzkent et al. have developed a framework for vehicle tracking application with hyperspectral data [12]. We also develop a Dynamic Data Driven Application Systems (DDDAS) for the hyperspectral video processing using the dataset with [1] and observed improved accuracy and performance by applying the dynamical band subset selection [6]. In that work, we optimized the accuracy with the constraints on execution time. As extended work, we implement this system to a embedded system and take the energy consumption constraint into account in this paper, since the energy consumption is one key consideration to evaluate the application on potable device.

This hyperspectral application focus on integrating LDspectral application to a embedded system and investigate the trade-off optimization between accuracy and real-time performance in hyperspectral video processing systems. In the meanwhile, we put emphasis on supporting flexible optimization involving the subset of available hyperspectral bands that is processed. The average execution time on average of each frame and power consumption are evaluated regarding to the selected bands.

IV. Background

A. Dataflow Design Tools

1) DICE: DICE stands for DSPCAD Integrative Command Line Environment, it is a package of utilities that facilitates efficient management of software projects. It

provides cross-platform support for model-based design methodologies and projects that integrate heterogeneous programming languages, as well as support for various design and testing methods. DICE can serve as a foundation for developing experimental research software in the area of digital signal processing systems. DICE can be used on multiple platforms, such as Linux, Mac OS, and Windows(with Cygwin equipped) [2].

2) LIDE: LIDE stands for the Lightweight Dataflow Environment, it is a lightweight design environment that allows designers to experiment with dataflow-based approaches for design and implementation of digital signal processing systems. Lightweight means that the programming model is designed to be minimally intrusive on existing design processes, and require minimal dependence on specialized tools or libraries [8]. LIDE includes application programming interfaces (APIs) for developing actors and edges in signal processing dataflow graphs. These APIs are not language specific, meaning that they are defined based on fundamental dataflow principles. Examples of language that can be used to programming in LIDE are: C, C++, MATLAB, Verilog, OpenCL, and CUDA [5].

B. Application Design Tools

In this section we introduce the tools and environment used for the Android-based hyperspectral video processing functionality. These tools consist of building block for the design and implementation of the hyperspectral video processing.

The proposed application involved using of the following tools: Android Native Development Kit (NDK), OpenCV4AndroidSDK, DICE, and LIDE. NDK allows the use of native code, such as C/C++ in development process of Android application. It also supports the compilation of native code on different architecture of mobile devides, such as ARM, MIPS, and X86, along with Android Debug Bridge (adb) shell, we can run the executable and avoid adding overhead from Java code which does not bring any difference in the functionality of the application.

1) Native Development Kit and Android Debug Bridge: The Android Native Development Kit (NDK) is a toolset that allows the use of C and C++ code on Android platform, it is especially convenient when porting available C/C++ code developed on other platform to Android platform [10]. NDK is also ideal when building computationally intensive application on Android platform, which provides extra performance compared to application developed in Java.

NDK allows user to build static, shared libraries and executable. Libraries can then be used either in the top-level Java code or towards building executable with NDK. Executable built with NDK can be executed using the Android Debug Bridge (ADB). ADB itself contains three components: client, for sending commands from local machine, daemon, which runs the commands sent from development machine in the background, and thirdly, server, which handles the communication between client and daemon that runs on the development machine [9]. Overall, NDK and ADB allows developers to migrate their existing native code/libraries to the Android platform and run it without a complete overhaul of their project.

2) OpenCV4Android SDK: OpenCV4Android SDK is a package provided by Open Source Computer Vision Library (OpenCV) and supports C++ and Java development specifically on Android platform [11]. In our experiment we use the C++ prebuilt static libraries that come with the OpenCV4Android SDK to build the static library for our driver and later use it towards driver executable.

C. Network Simulator

NS3 is an open source platform for simulating communication protocols which include the useful modeling for different network structure. It can be interfaced with external libraries and tools. The wireless network will be constructed using the spacing parameter with each node represent one communication node in the network. For example, in the factory each station node in the NS3 simulator could represent one rail, machine, machine controller, part generator or product store.

The co-simulator is the integration of the factory simulator in LiDE and the wireless network simulator in NS3. The communication events generated in LiDE simulator will be sent to NS3 and NS3 will simulate the communication using the specified network configuration. The network configuration is determined by the controllable parameters. Fig. 3 shows the LiDE/NS3 co-simulator block diagram. After NS3 finish the simulation of current event in the scheduler, it will send the feedback to LiDE simulator which will trigger the consequent simulation events.

V. Case Study

A. Simple Factory

1) Simple Factory modeling: In this research, the simulation of networked embedded factory is considered as an example. The factory consists of machine, rail, machine controller, part generator and product store, which are equipped with sensors and actuators. The sensor will detect the changes of machine or rail and send the corresponding message to machine controller; the controller will send commands back to the actuators according to current state and received message. All the communication are finished by the wireless network set up in the factory. The cosimulator we developed will simulate the communication events in the factory and evaluate the reliability and latency of the communication system regarding to different configuration and parameters.

Figure 1 illustrates a simple factory system that we use to demonstrate the dataflow-based modeling process in factory simulator. The factory processes parts that are generated by a Parts Generator. In the simulator the parts will be generated with some specified time interval. The time interval is modeled as some constant plus several random deviations. After the part is exported from the generator, it will be processed by a pipeline that consists of several distinct machines, for example, there are three machines in 1. The machine will take some time to work on the part, such as adding some specific feature to the part. After being processed by all the machine in the pipeline, the fully-processed parts will be sent to the subsequent product Store. The

processing time required by the machine and the transition time of part on the rail are also modeled as some constant number plus random deviation.

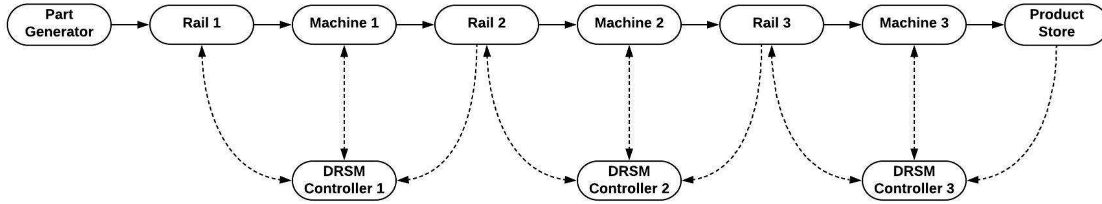

Fig. 1: A simple factory system.

The adjacent machines and rails in Figure 1 are controlled by a controller subsystem, which we refer to as machine controller. The three lower blocks in the figure represent the machine controller. We call them Dual-Rail, Single Machine (*DRSM*) controller, since the machine controller is designed to interface with two rails and a single machine. For example, the DRSM controller 1 will interface with Machine 1, Rail 1 and Rail 2, e.g. it sends a command to Machine 1 and it also sends command to Rail 1. At the same time, Machine 1, Rail 1 and Rail 2 will send message back to machine controller 1 to update their status. When a machine or rail completes a command that is sent from an associated machine controller, it sends an acknowledgment message back to that controller indicating the completion of the command operation. Then the machine controller could update the consequent commands message. Additionally, the state changes result from finishing command within the machines and rails will trigger corresponding notification messages to inform the machine controllers. All of these communication between machines and rails and their associated machine controllers is assumed to be carried out using wireless connections, which are shown as the dashed edges in Figure 1.

Our simulation system incorporate the capability of varying the factory size. This enhance the system with scalability both in the factory event simulation and wireless communication network design. The design of scalability involve the design of dataflow graph with factory size changes, wireless network connection and the

interface for scalability modeling. Each factory model simulated in our experiments consists of one or more pipelines of the form illustrated in Figure 1. For the experiment, each factory model has two size-related parameters — the number of pipelines N_p, and the number of machines per pipeline N_m, for example, the factory shown in Figure 1 corresponds to $N_p = 1$ and $N_m = 3$.

2) Factory Simulator Package: Based on the discrete communication characteristics and the dataflow modeling for the factory events, we developed the factory communication simulation package for the wireless communication of simple factory. This package includes the factory simulator in LIDE, the communication simulation package in NS-3 and the interface for the events transferring between LIDE and NS-3.

In our simulation framework, we incorporate an extension to LIDE for managing time because the dataflow model of computation, which LIDE is based on, is an untimed model with no built-in concept of time or time stamps. The “time-extended” version of LIDE that we employ is referred to as $\tau\mathrm{LIDE}$, where τ represents the incorporated notion of time, and we refer to the new factory simulation framework as $\tau\mathrm{LIDE}$–factorysim, which we abbreviate as TLFS (Tau Lide Factory Sim). The TLFS framework can be viewed as a design tool that applies timed dataflow concepts in novel ways to enable model-based cosimulation of factory process flows together with discrete-event simulation of communication networks that link physically-separated subsystems within the factories. [4]

The factory system is modeled as a dataflow graph in which actors represent distinct physical or computational components within the factory, and edges represent the flow of messages or physical entities. Messages include the commands from the machine controllers and the state update from machines and rails, whereas the physical entities are parts for products that are being manufactured. The actors include actors of individual machines within a factory, individual rails that connect different machines, and machine controllers that send signals to machines.

Figure 2 illustrates the architecture of TLFS. The top blue-, middle green-, and

bottom red-colored parts of the diagram correspond to parts that pertain to factory process flow simulation, interfacing between the factory and network simulation subsystems, and communication network simulation.

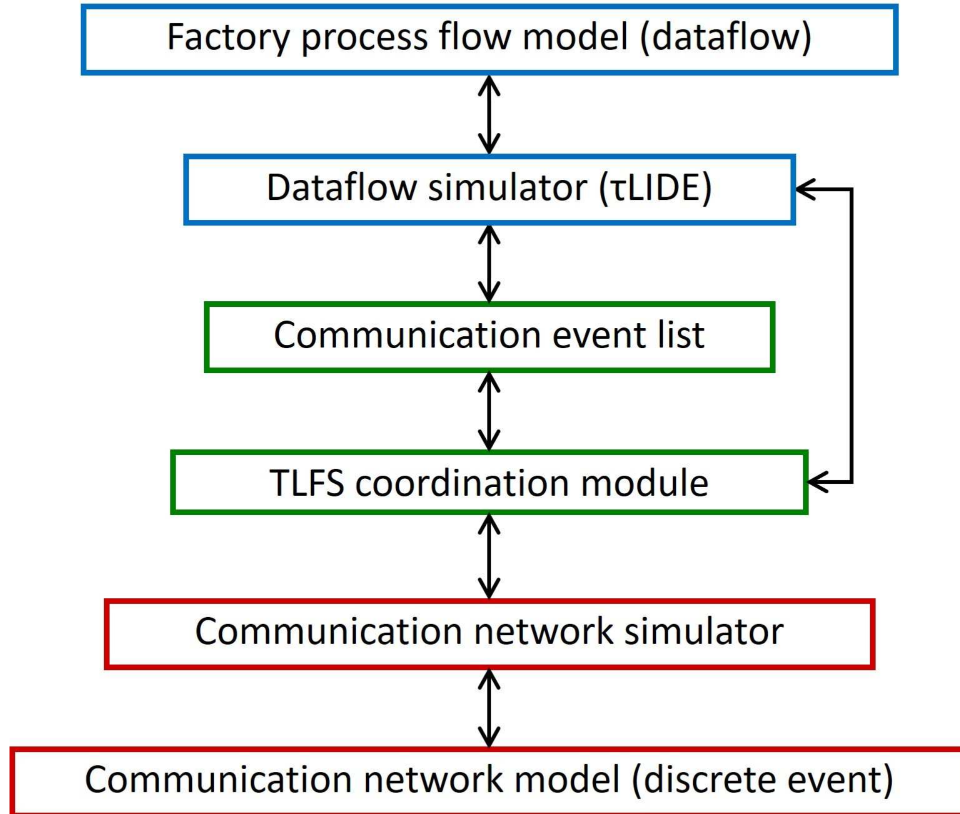

Fig. 2: Cosimulation architecture based on TLFS.

As illustrated in Figure 2, coordination module plays the role of transferring the relevant simulation events between the factory and network simulation subsystems. The events could be classified into two groups — send events and receive events. Send events refers to the events that trigger event that are generated in the factory simulation. Receive events are generated in the communication network simulator. These events are transferred between simulation subsystems using a dedicated event list, which are named as "communication event list". The coordination module acts as a gateway to achieve proper synchronization of communication events.

3) Coordination Module: Algorithm 1 shows a pseudocode sketch of the coordination module. Here, `ntime` represents the value of simulated time in the CNS (*network time*), and `nresult` is a simple data structure that is used to communicate selected status information from the CNS tool to the coordination module. In particular, `nresult.result_time` gives the current value of network time, and `nresult.receive_events` provides the list of network receive events that have been generated in the CNS tool during the most recent call to the CNS tool. The function `save_receive_events` removes events from `nresult.receive_events`, and generates corresponding events in τLIDE that trigger processing of the received data (see Figure 1). The dataflow simulator has access to the communication event list directly through special *communication interface actors* within the factory process flow model.

Algorithm 1

```
parameter T_N: simulated time in network simulator
parameter T_L: the maximum time limit for the simulation
parameter T_E: the time for next event in the list
parameter deadlock: flag to indicate whether deadlock happens
while T_N < T_L and !deadlock do
    T_E = next_lide_event_time_network()
    simulate_network(T_E, nresult)
    if !is_empty(nresult.receive_events) then
        save_reveive_events
    ntime = nresult.time
    if !deadlock then
        simulate_lide(ntime, lide_result)
        transfer_comm_events(comm_event_list)
```

The `next_lide_event_time` function is a method of τLIDE that returns the time of the earliest pending event within the event list of the dataflow (factory) simulation. The `simulate_network` function serves as a "wrapper" for whatever CNS tool is being used in the given application of TLFS. This function uses application programming interfaces of the CNS tool to simulate the FA system's communication network until a new receive event

is generated or the `next_lide_event_time` is reached, whichever happens first. If (a) the return value from `next_lide_event_time` is infinite, which means that there are no pending dataflow events, and (b) there are no more pending communication events in NS-3, then `simulate_network` stops with `network_result.deadlock = true`.

The simulation (`while` loop in Figure 1) stops when this deadlock condition is reached or when the predefined simulation time limit `time_limit` is reached. If the `next_lide_event_time` is infinite (which means theres no event in τLIDE event list) and theres also no event in the third party simulator, then deadlock condition holds. Each time τLIDE is called (using function `simulate_lide`), it updates the event list of LIDE simulator. The pending event with smallest time (`next_lide_event_time`) is used as a stop point for the third party simulator. If there is a communication event generated by LIDE simulator, we transfer that event to the third party simulator.

4) τLIDE: τLIDE is a simulation framework to support the assessment and experimentation of dataflow graph with the need of chronological simulation. It contains its own timeline and also be able to incorporate process time for each actor. τLIDE allows the untimed LIDE model to co-simulate with other timed simulator, which is useful for examples like communication network simulation, neural network simulation, etc. The Global Dataflow Time (GDT) is the wall clock handled by τLIDE simulator, which represents the time flow of dataflow graph. Its the key component that changed untimed LIDE into timed structure. An event is generated in τLIDE when some actions should be taken when executing LIDE graph. For example, an actor needs to be fired, an message need to be printed, a flag or some data required to be transferred with other coordinate simulator, etc. Firing completion event is referred to the event action corresponds to actor firing. When an actor is enable firing, instead of firing that actor directly, τLIDE generates a corresponding firing completion event based on the latency of that actor.

The CNS tool is responsible for simulating latencies associated with sending and receiving data across the communication network. In general, CNS tools take into account channel characteristics, network traffic conditions, and transmitter-receiver separation (distance) in determining these latencies. Firing completion event is referred to the event action corresponds to actor firing. When an actor is enable firing, instead of firing that actor directly, τLIDE generates a corresponding firing completion event based on the latency of that actor. In general, each dataflow actor A in a TLFS factory process flow model has an associated execution time estimation function, which is denoted by θ_A or by θ if A is understood from context. The arguments to the function include any parameters and state variables of A.

When a new firing of A becomes enabled at some simulated time t, TLFS calls the θ_A function to determine the amount of simulated time T_f that will be expended by the firing. Here, by an *enabled* firing, we mean a firing for which there is sufficient data on the input edges of the associated actor A, as defined by the consumption rate specification for the next mode of A.

Upon determining the value T_f, TLFS schedules a *firing completion* event to be processed by the simulator at time $(t + T_f)$. The firing completion event triggers the execution of a τLIDE function that carries out a single firing of A, which in turn updates the input and output edges (FIFOs) of A based on the token consumption and token production that occurs as part of the firing.

In τLIDE, the actors can be fired only if GDT has reached the scheduled time of that actor. Once an actor is enabled firing, τLIDE schedule an event for that actor with time $GDT + T_f$, and wait for invocation after T_f time later. If τLIDE finds that all the pending events have scheduled time greater than ntime, then this round of LIDE simulation stops and return the result to the upper level module. When `simulate_lide` function is called for a given `n_time`, τLIDE simulator set the flag done to false and start simulating until this flag becomes true. Then it will check every actors in LIDE graph. If theres an EAU (Enabled And Unscheduled) actor (actor

that is enabled such that scheduled[A] = false), calculate associated execution time by estimation function f_A according to the latency model, and then schedule the event in the event list. Then it check whether the `next_lide_event_time` (the time of the earliest pending event within the event list) is smaller or equal to ntime. If yes, advance GDT to that time and fire the correspond actor. If not, advance GDT to ntime and jump out of the τLIDE simulation.

5) Communication Interface Actors: Communication interface actor is a special type of actor which is responsible for exchanging data with the third party simulator. Event that goes to the third party simulator may done by the send interface actor, and similarly, the feedback event from the third party simulator can be received by LIDE through receive interface actor. Events generated by these two actor are referred to communication events. Communication interface actors are used in factory process flow models to represent functionality for sending or receiving data across wireless communication channels. These actors are used to provide modular interfaces between the dataflow subsystem within a TLFS simulation model and the TLFS coordination module (see Figure 2), which is used to provide time synchronization and information transfer between the dataflow simulator and the CNS tool.

In TLFS, we use two different types of communication interface actors for sending and receiving data across the wireless network. These are referred to, respectively, as *send interface actors* (*SIAs*) and *receive interface actors* (*RIAs*). When an SIA fires (executes) in the dataflow simulation, it inserts one or more events into the communication event list. This list is used by the TLFS coordination module to transfer events between the dataflow simulator and the CNS tool.

Similarly, events associated with the reception of data in the CNS tool (network receive events) trigger the firing of RIAs in the dataflow simulation. This triggering is enabled again by the coordination module, which injects an event into the τLIDE event list corresponding to each network receive event that it detects. This process of injecting reception-related events into τLIDE is represented by the function call

labeled `save_receive_events` in Figure 1. An RIA firing completion event is scheduled in τLIDE for each packet reception. The time of the event is the receive time as determined by the CNS tool.

Note that there need not be a one-to-one correspondence between the SIAs and RIAs in an TLFS model. The routing of packets between SIAs and RIAs is achieved through information in the packets (as they are assembled in the dataflow simulation), and the communication protocols that are being used (as they are simulated in the CNS tool).

6) Factory Simulator Package Usage: In this section, we show the features and usage of factory simulator package, which consist of the τLIDE package and CNS package. We will show the APIs, the usage of both τLIDE and the testing method for the co-simulator.

The τLIDE package incorporates the timed simulation for discrete events of the factory in dataflow environment. Useful functions and abstract data type are defined in τLIDE to support the timed dataflow graph.

Program 1 Prototype for new function of τLIDE.

```
taulide_c_sim_context_type *taulide_c_sim_new();
```

Program 2 Prototype for graph set function of τLIDE.

```
void taulide_c_sim_set_graph(taulide_c_sim_context_type *context,
        lide_c_graph_context_type *graph);
```

Program 3 Prototype for actor execution time set function.

```
void taulide_c_sim_set_fa(taulide_c_sim_context_type *context,
        int actor_id, taulide_c_exec_time_function_type fp);
```

Program 4 Prototype for actor reset function.

```
void taulide_c_sim_set_ra(taulide_c_sim_context_type *context,
        int actor_id, taulide_c_reset_function_type fp);
```

Program 5 Prototype for validity check function.

```
boolean taulide_c_sim_check_validity(taulide_c_sim_context_type
        *context);
```

Program 6 Prototype for simulation run function.

```
double taulide_c_sim_simulate(taulide_c_sim_context_type *context,
        double expire_time, double end_time);
```

Program 7 Prototype for simulation reset function.

```
void taulide_c_sim_reset(taulide_c_sim_context_type *context);
```

Program 8 Prototype for add event function.

```
boolean taulide_c_sim_add_event(taulide_c_sim_context_type *context,
        int actor_id, int event_type, double time);
```

Program 9 Prototype for get GDT function.

```
double taulide_c_sim_get_gdt(taulide_c_sim_context_type *context);
```

Program 10 Prototype for simulation free function.

```
void taulide_c_sim_free(taulide_c_sim_context_type *context);
```

The taulide_c_sim_new function will create and allocate a new factory simulation context, and the return type is taulide_c_sim_context_type. All other programming function will work on this context. After creating a new context, we can set it with the taulide_c_sim_set_graph function, which takes LIDE graph as one input parameter. The input graph argument contains the information of the factory, including the actors and edges graph elements. In τLIDE, each actor has an execution time estimation function associated with it. The taulide_c_sim_set_fa could set the execution time of each mode in the actor, which is indicated by the actor_id parameter. Similarly, the taulide_c_sim_set_ra function will reset the execution time. The taulide_c_sim_check_validity function checks whether the graph and actor execution time have been set, and it will return true if they are already set.

After all the graph and actors execution time have been set, we can run taulide_c_sim_simulate function to start the running of the simulation. It simulates the current graph for some amount of time, starting from where the previous call ends. The taulide_c_sim_next_event and taulide_c_sim_next_event_time functions will return the event with smallest time (next event) and its timestamp from the event list, respectively. This information help the simulator driver to find the next_lide_ever_time and offer debug information to check the simulation status during the simulation. The taulide_c_sim_reset resets the simulation context to initial status. After the simulation completion, we can terminate and free the simulation context using taulide_c_sim_free function.

Program 11 Data structure for the factory event.

```
struct factory_event {
        char sender;
        int sender_id;
        char receiver;
        int receiver_id
        char message;
        double time;
};
```

The Program 11 shows the data structure for the communication events, which contains necessary information for determining the source and destination of the communication message and the message content. It also contains the timestamp of this event. The communication interface actor and the event scheduler will use this useful information in the coordination of the events.

Program 12 Prototype for the network parameter setup function.

```
std::vector <Ptr<MyApp> > network_setup (NetworkPara &net_config);
```

The `network_setup` function set the parameters for the network with the `net_config`, which has the type of `NetworkPara`. The `NetworkPara` is a class we defined to handle all the network parameters including the parameters of the factory size, the type of protocol, the propagation loss model and the antenna TxGain. The `network_setup` function returns a vector of `MyApp`, which is the application we define for all the communication events. Each application contains the necessary packet sending functions, including the socket setup function, the application start and stop function and the packet send function.

Program 13 Prototype for the event scheduler function.

```
void ScheduleTx (std::vector <Ptr<MyApp> > app,
        struct factory_event *comm_event);
```

The `ScheduleTx` function offers the interface to schedule communication event to the simulator scheduler. It takes the application vector and factory event data as input. The factory event data contains the source/destination information, the message and the timestamp, which could help the scheduler to find the corresponding application and insert it into the corresponding time to the discrete event scheduler.

Program 14 Prototype for the simulator run function.

```
struct factory_event *simulator_run (std::vector <Ptr<MyApp> > app,
        double lide_result_time);
```

After all the network parameters setup and event scheduler are finished, the `simulator_run` function could run the simulator. The `simulator_run` function takes the vector of application and `lide_result_time` as input parameters. It will simulate till the `lide_result_time` and pend to wait for next event being scheduled. This function returns the data with type of `factory_event`. The timestamp in this data structure is set to be the time when the packet is received successfully by the receiver side. The RIA in τLIDEsimulator will use this timestamp as reference to trigger following events. The timestamp in the received events are also useful in the calculation of the communication delay in the packet transmission.

To run the simulator, the user should run the factory simulator driver executable program with two input file arguments as follow example. All the necessary parameters are stored in these two files. The `in_parameter.txt` contains the parameters related to the factory attributes including the number of parts to be processed, the time interval of the part generation, the time of each machine working on the part and the time needed by the rail to transit the part. The `net_parameter.txt`

are the parameters for the network configuration, such as the type of protocol, the parameters in propagation loss model, the antenna TxGain and so on. The output of the program will prompt all the communication events and the packet transimision stastics including the total counts of packets, the total communication delay and the packet retransmission attempts.

Example 1 Example to run the simulator.

```
lide_c_factorysim_driver.exe in_parameter.txt net_parameter.txt
```

B. Hyperspectral Image Processing

In this section we introduce our multispectral video processing application targeted for Android platform, as shown in Figure 3. This application illustrate the feasibility of implementing band subset selection (BSS) on a mobile platform, under performance and bettery constraint. In this application, the available multispectral data comes from a set $Z = \{B_1, B_2, \ldots, B_N\}$ of spectral bands, where N denotes the total number of available bands. Under performance constraints it is not possible to perform processing on all 2^Z bands, where 2^Z is the power set of Z, equals to all subsets of Z.

The problem is to choose a subset S of 2^Z so that processing of this selected subset of bands can maximize video analysis accuracy subject to performance constraint C_r, where C_r is the constraint on execution time performance for a particular video processing task, with units of time [6]. Band subset selection is utilized and invoked at time interval determined by the reconfiguration interval parameter T_r. Band subset selection tries to optimize the subset of bands that is to be processed during the interval of video processing. The output of BSS is a vector $S = \{B_{s1}, B_{s2} \ldots B_{sm}\}$ which is a subset of Z. This vector is later used in the video processing.

LDSpectral is demonstrated through a case study involved with background subtraction. We focus on three aspects of the experiment: $F_{measure}$, shows how accurate

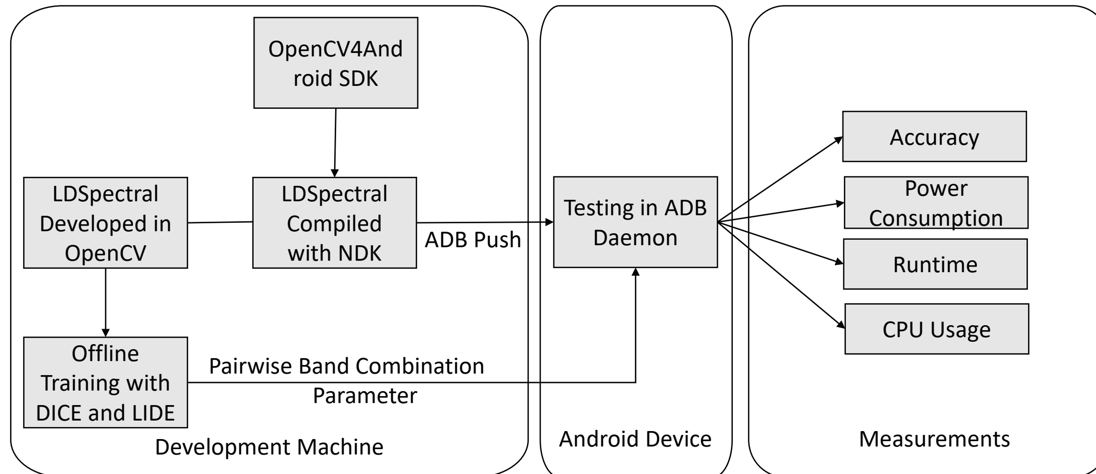

Fig. 3: Block diagram of LDSpectral Android version development

the foreground extraction is; Average execution time t_{ave}, the average time needed to extract foreground; And power consumption W_{pwr} during one iteration of experiment.

LDSpectral was originally developed in LIDE and C++ on PC to implement band subset processing dataflow subsystem. The dataflow subsystem consists of seven actors as shown in Figure 4. The image read actor takes input images by pointing a stream of pointers to the images so that they can be processed by the background subtraction individually. Upon outputting from image read actor, each image contains m components, where each component matches one of the element in the spectral bands(i.e. an element of the set S). The image combination actor performs pixel-level fusion for these images, therefore the selected bands component are combined into a single "fused" image.

The background subtraction actor extracts foreground and passes it as an image pointer at output to the foreground filter actor, which removes noise of previous actor's output by performing erosion and dilation operations. Then the foreground binarization actors takes the this output and converts it into binary form so that every pixel in the image is determined to be either foreground or background. The binarization is completed by thresholding input images with a user defined threshold to classify

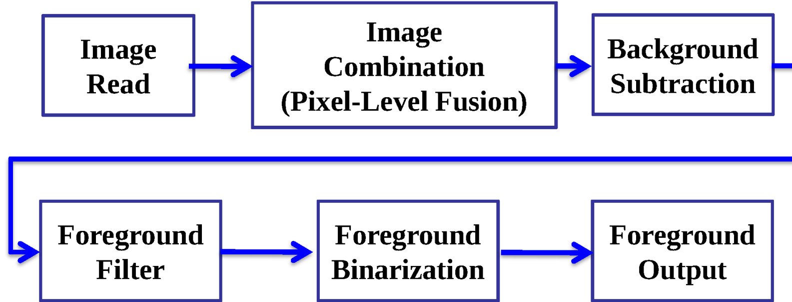

Fig. 4: Block diagram of band subset processing in the background subtraction system [6].

foreground whenever the pixel value exceeds the preset threshold. The purpose of this actor is to enhance accuracy of foreground detection and output the resulting binary image to the next actor in the dataflow graph. Finally we use the foreground output actor to store the classification result for each image in the specified output directory with index related to their corresponding input frames from the dataset.

We perform the training of the LDSpectral on a loptop equipped with an Intel i7-4710HQ CPU, 16GB RAM, and the Ubuntu 16.04 LTS operating system. Testing phase was performed on the Nexus 7 tablet with Nvidia Tegra 3 Quad-core 1.2 GHz Cortex-A9 CPU, 1GB of RAM and 16GB internal storage. The tablet is running on Android 5.1.1 operating system. The experiment results are discussed in the following section.

VI. Experiments

A. Factory Cosimulator

1) Simulation Parameters: Each factory model simulated in our experiments consists of one or more pipelines of the form illustrated in Figure 1 Each factory model has two size-related parameters — the number of pipelines N_p, and the number of machines per pipeline N_m. Thus, the example shown in Figure 1 corresponds to $N_p = 1$ and $N_m = 3$.

Table I summarizes the other key simulation parameters used in our experiments. A given data point in the experiments is derived by executing a simulation with the same settings N_s times, and averaging the results over the N_s executions. Each such simulation involves N_j generated parts for each Parts Generator actor in the factory dataflow graph. The simulation completes when all of the generated parts are fully processed in their respective pipelines. Since there is one Parts Generator actor per pipeline, this means that each simulation involves processing a total of $(N_p \times N_j)$ parts.

The values of t_m and t_r give, respectively, the estimated execution time values used in the simulation models for a machine to process a part, and for a rail r to move a part from one end of r to the other end. Similarly, t_i is the estimated time required to generate a new part after the previous part has been generated. The values of t_m, t_r, and t_i are used in the execution time estimation functions (θs) for the relevant actors (see Section V-A4).

TABLE I: Simulation parameters.

Parameter	Value
Parts Generated Per Pipeline N_j	100
Number of Simulation Iterations N_s	10
Machine Processing Time t_m	10 sec
Rail Transfer Time t_r	4 sec
Part Generation Interval t_i	10 sec
Channel Frequency	2.4 GHz
Large Scale Path Loss Model	Log-distance
Decay Exponent α	3
Distance Reference d_0	1 m
Loss at Reference L_0	46.6777 dB

The parameters α, d_0, and L_0 in Table I are related to the simulation of propagation path loss. In our simulations, we apply features in NS-3 for using the log-distance path loss model to estimate signal loss in communication channels. The log-distance

model is often used to estimate path loss within buildings. In this model, the power loss at the receiver side when transmitting over a distance d is calculated by

$$L = L_0 + 10\alpha log_{10}(\frac{d}{d_0}) + Z, \tag{1}$$

where L_0 is the path loss at the reference distance, d_0 is the reference distance, α is the decay exponent, and Z is the log-normal shadowing.

2) Experiments with Different Protocols: We first use TLFS to study the average communication delay T_c for a fixed factory size, and the variation in T_c for different communication protocols — in particular, for different variants of IEEE 802.11. Using TLFS, we measure T_c as the average time difference between the time when a communication packet P is successfully received (through an RIA), and the time when P was transmitted (through an SIA). This average is taken over all packet communications within a given simulation.

In this experiment, we use a factory model with a single pipeline that contains 3 machines — that is, $N_p = 1, N_m = 3$.

Figure 5 shows a box plot representation of how T_c was found to vary across four different protocols — IEEE 802.11xx, for $\mathrm{xx} \in \{\mathrm{ac}, \mathrm{b}, \mathrm{g}, \mathrm{n}\}$. Significant performance variation is shown between the best-performing protocol in this context (IEEE 802.11g) and the worst-performing one (IEEE 802.11b). One reason for the relatively low performance of IEEE 802.11b may be its low maximum data rate. Compared with other protocols, which can achieve 54 Mbps speed, the maximum speed of IEEE 802.11b is only 11 Mbps.

Overall, the results in Figure 5 show a clear advantage of IEEE 802.11g in terms of T_c for the factory model studied in this experiment.

3) Scalability Experiments: Next, we study how communication performance changes as we increase the factory size (N_p, N_m) and the distance between factory subsystems. Here we study five different factory dataflow graphs, denoted (a) through (e), with sizes that are defined, respectively, as $(N_p, N_m) =$

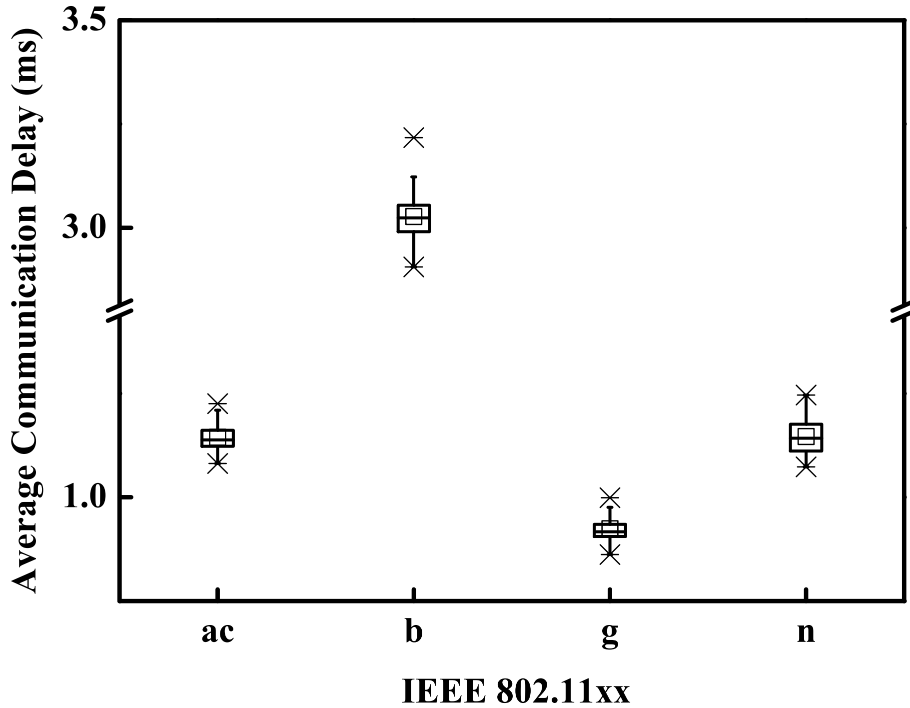

Fig. 5: A box plot representation of the variation in average communication delay across different communication protocols.

$(1,3), (1,4), (1,5), (4,4), (5,5)$. The total number of network nodes in the CNS tool modeling subsystem are, respectively, 11, 14, 17, 56, and 85. Each rail, machine, and machine controller corresponds to a distinct network node. Additionally, each Parts Generator and Parts Sink is modeled as a separate network node.

We define a *distance parameter* d associated with the simulations in this experiment. The units of this parameter are meters. For each factory pipeline, the spacing between adjacent network nodes is set to d. Additionally, for factory models that consist of multiple pipelines, the successive pipelines are spaced apart by distance d. Note that for the experiments reported in Section VI-A2, we used a constant distance value of $d = 10$.

Figure 6 and Figure 7 show the variation in average communication delay and packet

retransmission rate, respectively, for the five different factory sizes defined above, and for different settings of d for each factory size. In each of these two figures, each of the five plots corresponds to a distinct (N_p, N_m) pair, and each curve within a given plot corresponds to a distinct value of $d \in \{5, 10, 15, 20, 25\}$. The data is plotted for each of the four IEEE 802.11 variants discussed in Section VI-A2. In Figure 7, the vertical axis represents the fraction of packet transmissions that have to be repeated due to errors in the original transmission. For example, a value of 0.5 means that 50% of the packets have to be retransmitted. In addition to increasing communication delays, packet retransmissions result in energy consumption overhead due to the increased operational load placed on the communication transceivers in the system.

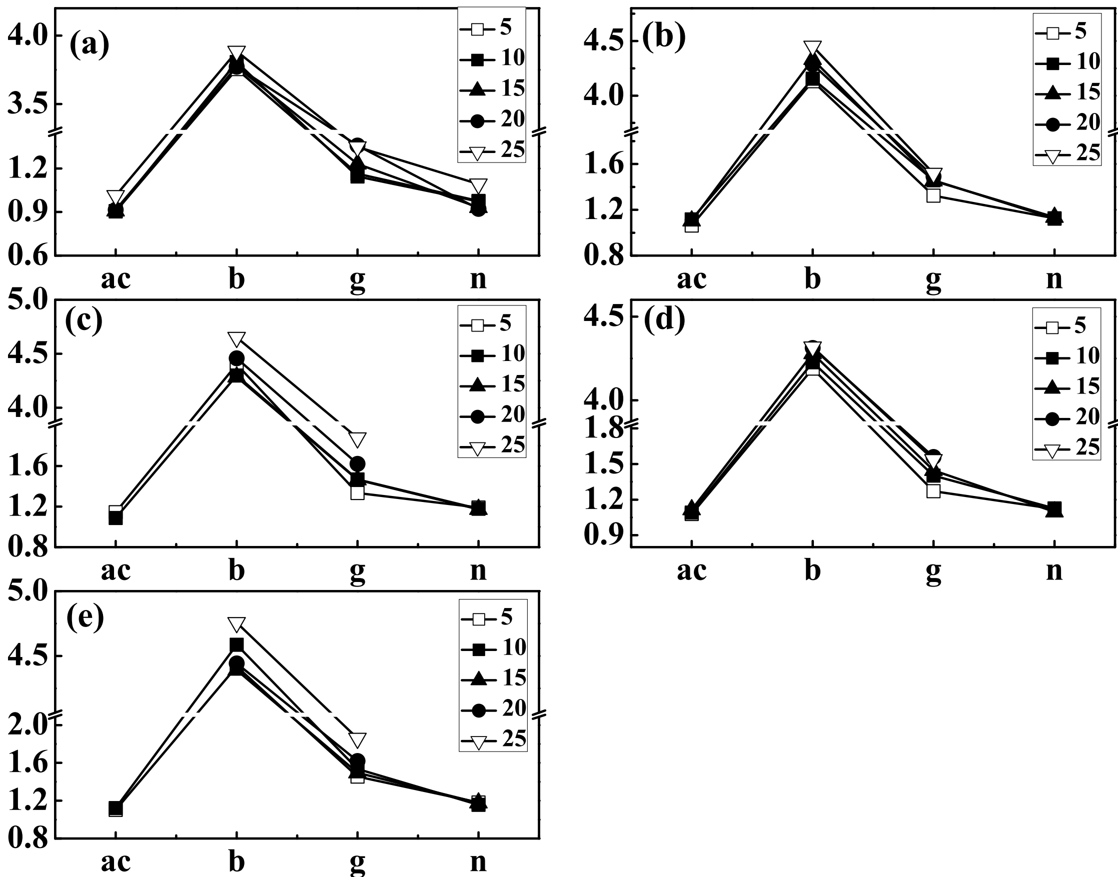

Fig. 6: Variation in average communication delay for different factory sizes, distance parameter settings, and IEEE 802.11 variants.

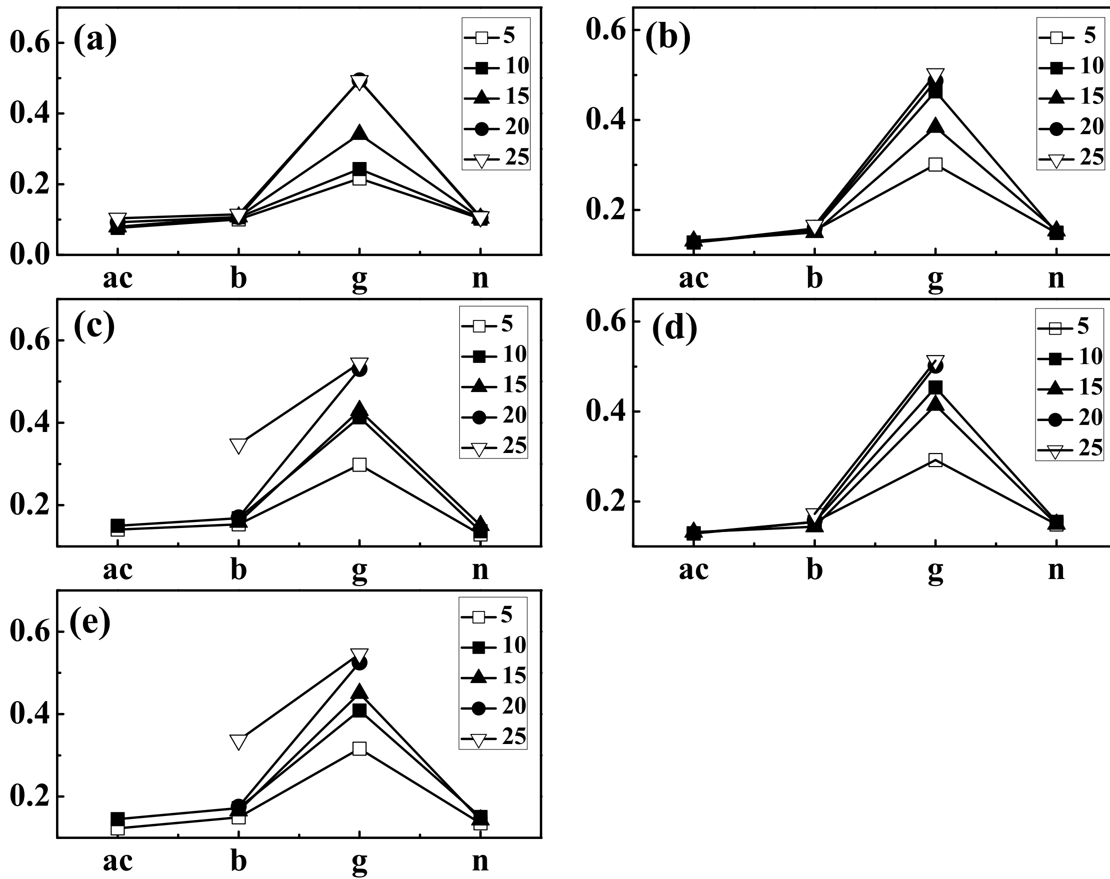

Fig. 7: Variation in packet retransmission rate for different factory sizes, distance parameter settings, and IEEE 802.11 variants.

The results in Figure 6 and Figure 7 show the general trends that one would expect of increasing communication delay and packet retransmission rate with increases in the distance parameter value d, and increases in the factory size. The results also provide insight into how different IEEE 802.11 protocol variants perform for the different factory size/distance combinations that are evaluated. The simulations carried out using TLFS provide a quantitative assessment of all of these trends, and the associated factory system design trade-offs. The results also help to validate the capabilities of TLFS and demonstrate these capabilities with further concreteness.

The results in II show the average communication delay with different antenna TxGain. The decreasing of time delay is observed with the increasing of TxGain in

TABLE II: Variation in average packet transmission delay (ms) for different antenna TxGain(dB) and IEEE802.11 variants.

	0	**2.5**	**5**	**7.5**	**10**	**12.5**	**15**	**17.5**	**20**
80211ac	**0.9159**	**0.9077**	**0.9042**	**0.8942**	**0.8458**	**0.9176**	**0.9248**	**0.9078**	**0.9148**
80211b	**3.3730**	**3.3440**	**3.3973**	**3.3056**	**3.3347**	**3.2826**	**3.3290**	**3.3346**	**3.2817**
80211g	**1.0917**	**1.0648**	**1.0613**	**1.0577**	**1.0586**	**1.0409**	**1.0572**	**1.0467**	**1.0278**
80211n	**0.9483**	**0.9310**	**0.9286**	**0.9284**	**0.9228**	**0.9355**	**0.9455**	**0.9474**	**0.9241**

some range as expected. However, keeping increasing the TxGain will increase the time delay when the TxGain is greater than some value. This could be explained by the interference introduced by the antenna with large power.

B. Hyperspectral Image Processing

During training stage of our experiment, we follow the procedure in [6] to select the pairwise band combination (PBC) parameter α. The dataset we use consists of 1102 images, each of these images contains 7 spectral bands. 6 of the spectral bands are visible bands, meaning they are visible to human. The last band is near-infrared band. The dataset is divided into two part: 735 images for training and 367 images for testing.

The purpose of training is to optimize the performance of each two-band subset. For a band subset $\{b_{s1}, b_{s2}\}$, we use the following equation to derive the PBC parameter α,

$$y = \alpha \times x_1 + (1 - \alpha)x_2 \tag{2}$$

In the above equation, x_1 and x_2 stand for corresponding pixel values for images from two-band subset at same coordinate (a, b), α is the relative weight parameter and $\alpha \in [0, 1]$, and y is the pixel fusion result at coordinate (a, b). We performed a exhaustive search for $\alpha \in \{0, 0.1, 0.2, \ldots, 1\}$ and then select a PBC that outputs the highest accuracy for subsystem 4. During testing stage we use the selected value to test for accuracy with same two-band subset.

Accuracy is measured in harmonic mean for background subtraction accuracy, similiar to [1]. The harmonic mean is defined as follows,

$$F_{measure} = 2 \times \frac{recall \times precision}{recall + precision} \tag{3}$$

And the definition for $recall$ and $precision$ are as follows

$$precision = \frac{n_c}{n_f}, \text{and } recall = \frac{n_c}{n_g} \tag{4}$$

where n_c stands for correctly classified foreground pixels, n_j stands for number of pixels classified as foreground, and n_g is the number of foreground pixels in the ground truth.

During training stage of our experiment, we follow the procedure in [6] to select the pairwise band combination (PBC) parameter α. The dataset we use consists of 1102 images, each of these images contains 7 spectral bands. 6 of the spectral bands are visible bands, meaning they are visible to human. The last band is near-infrared band. The dataset is divided into two part: 735 images for training and 367 images for testing.

The purpose of training is to optimize the performance of each two-band subset. For a band subset $\{b_{s1}, b_{s2}\}$, we use the following equation to derive the PBC parameter α,

$$y = \alpha \times x_1 + (1 - \alpha)x_2 \tag{5}$$

In the above equation, x_1 and x_2 stand for corresponding pixel values for images from two-band subset at same coordinate (a, b), α is the relative weight parameter and $\alpha \in [0, 1]$, and y is the pixel fusion result at coordinate (a, b). We performed a exhaustive search for $\alpha \in \{0, 0.1, 0.2, \ldots, 1\}$ and then select a PBC that outputs the highest accuracy for subsystem 4. During testing stage we use the selected value to test for accuracy with same two-band subset.

Accuracy is measured in harmonic mean for background subtraction accuracy,

TABLE III: Derived energy consumption, execution time, and CPU usage on Android device.

N_B	7	14	28	56
C_e (mAh)	16.20	22.18	33.65	63.94
C_t (s)	94.16	133.88	209.74	397.22
C_u (%)	32.18	28.05	28.41	26.55

similiar to [1]. The harmonic mean is defined as follows,

$$F_{measure} = 2 \times \frac{recall \times precision}{recall + precision} \tag{6}$$

And the definition for $recall$ and $precision$ are as follows

$$precision = \frac{n_c}{n_f}, \text{and } recall = \frac{n_c}{n_g} \tag{7}$$

where n_c stands for correctly classified foreground pixels, n_j stands for number of pixels classified as foreground, and n_g is the number of foreground pixels in the ground truth.

Testing stage is performed on the Nexus 7 tablet's command-line environment. The testing device is wireslessly connected to host computer without external power supply. PBC parameters from training stage are predefined in the program. Energy consumption is measured through ADB's battery information (i.e. current, voltage) along with CPU usage during testing. These information is used to calculate power consumption and CPU usage so that we could demonstrate the feasibility of running LDSpectral on mobile device.

The dataset we used is the DIRSIG's synthetic aerial hyperspectral cideo for vehicle tracking. The dataset contains 110 frames of video recording, each frame consists 61 spectral bands of image. The ground truth of vehicle's foreground was manually extracted since it was not provided in the dataset. We select 7, 14, 28, and 56 as band numbers N_B for testing on the dataset to perform background subtraction. For

each band number, we perform the testing three times and collect average current draw, runtime, and CPU usage. Then from these data we calculate the average energy consumed for each band number C_e in mAh, average runtime C_t in s, and average CPU usage C_u in $\%$.

From table III we can see that the average energy consumed C_e and average runtime C_t of our background subtraction application on Android is positively correlated to the band number N_B. The average CPU usage C_u, however, remains stable as the number of threads during execution was limited to 4 and the change in N_B has no effect on CPU usage, which always remains around 25% to 30%.

VII. Conclusion

The embedded networked cosimulation system is developed in dataflow modeling with discrete event network simulator. The simulation result from this work shows an efficient way for integrating dataflow models of embedded processing with state-of-the-art techniques for modeling and simulation of computer networks. The results with different protocol type and network parameters show the possibility of multiobjective system level optimization of the network design. The cosimulation could help in the design of networked embedded systems that have improved trade-offs among reliability and efficiency. Networked embedded systems for applications such as surveillance and remote sensing could also benefit our application of distributed hyperspectral image processing system.

References

[1] Y. Benezeth, D. Sidibé, and J. B. Thomas. Background subtraction with multispectral video sequences. In *Proceedings of the Workshop on Non-classical Cameras, Camera Networks and Omnidirectional Vision*, 2014.

[2] S. S. Bhattacharyya, W. Plishker, C. Shen, N. Sane, and G. Zaki. The DSPCAD integrative command line environment: Introduction to DICE version 1.1. Technical report, Institute for Advanced Computer Studies, University of Maryland at College Park, October 2011.

[3] L.-J. Ferrato and K. W. Forsythe. Comparing hyperspectral and multispectral imagery for land classification of the lower Don River, Toronto. *Journal of Geography and Geology*, 5(1):92–107, 2013.

[4] J. Geng, H. Li, Y. Liu, Y. Liu, M. Kashef, R. Candell, and S. S. Bhattacharyya. Model-based cosimulation for industrial wireless networks. In *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, Imperia, Italy, June 2018.

[5] K. Lee, H. Ben Salem, T. Damarla, W. Stechele, and S. S. Bhattacharyya. Prototyping real-time tracking systems on mobile devices. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 301–308, Como, Italy, May 2016.

[6] H. Li, K. Sudusinghe, Y. Liu, J. Yoon, M. V. D. Schaar, E. Blasch, and S. S. Bhattacharyya. Dynamic, data-driven processing of multispectral video streams. *IEEE Aerospace and Electronic Systems Magazine*, 32(7):50–57, 2017.

[7] Y. Liu, R. Candell, K. Lee, and N. Moayeri. A simulation framework for industrial wireless networks and process control systems. In *IEEE World Conference on Factory Communication Systems*, pages 1–11, 2016.

[8] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pages 640–645, Washington DC, USA, November 2010.

[9] The ADB Team. Android Debug Bridge guide. https://developer.android.com/studio/command-line/adb, 2018. Online; accessed 26 June 2018.

[10] The NDK Team. Native Development Kit guide. https://developer.android.com/ndk/guides, 2018. Online; accessed 26 June 2018.

[11] The OpenCV Team. OpenCV4Android SDK tutorial. https://opencv.org/platforms/android/, 2018. Online; accessed 26 June 2018.

[12] B. Uzkent, M. J. Hoffman, and A. Vodacek. Integrating hyperspectral likelihoods in a multidimensional assignment algorithm for aerial vehicle tracking. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(9):4325–4333, 2016.

[13] S. Won, C. Shen, and S. S. Bhattacharyya. NT-SIM: A co-simulator for networked signal processing applications. In *Proceedings of the European Signal Processing Conference*, pages 1094–1098, Bucharest, Romania, August 2012.