CSense: A Stream-Processing Toolkit for Robust and High-rate Mobile Health Systems

Farley Lai, Syed Shabih Hasan, Austin Laugesen, Octav Chipara
University of Iowa

***Abstract*—This paper presents CSense – a stream-processing toolkit for developing robust and high-rate mHealth systems in Java. CSense addresses the needs of these systems by providing a new programming model that supports flexible application configuration, a high-level concurrency model, memory management, and compiler analyses and optimizations. The compiler analyses detect a range of programming errors including application composition errors, improper use of memory management, and data races. We identify that memory management and concurrency limit the scalability of stream processing systems. We incorporate memory pools, frame conversion optimizations, and custom synchronization primitives to develop a scalable run-time. CSense is evaluated on Galaxy Nexus phones running Android. Empirical results indicate that our run-time achieves 25 times higher steam processing rate compared to a realistic baseline implementation. Moreover, our frame analysis optimizes the exchange of data between components to achieve an additional 45% improvement in stream rate. We demonstrate the versatility of CSense by developing three mHealth systems.**

I. Introduction

Mobile health (mHealth) systems are expected to transform how healthcare professionals collect information about patients including information regarding a patient's physiology, physical activities, and social interactions. A typical mHealth system collects readings from sensors, extracts domain-specific features from the readings, and computes higher-level inferences and representations of a patient's activity and physiological state based on the computed features. Results of recent studies on mHealth systems have shown the feasibility of collecting medical records with higher resolution than would be possible through manual data collection methods [1], [2]. However, experience has also shown that the development of mHealth systems is particularly time demanding and challenging even for expert programmers. The prolonged development time has hindered the advancement of mHealth systems as a majority of the time is spent ensuring that the system operates robustly within the resource constraints of sensor platforms rather than exploring different sensing trade-offs or novel system architectures.

A key challenge to mHealth systems is to process tens of thousands of sensor readings in real-time or within a few seconds after their collection. This is particularly challenging due to the limited computational resources and energy budgets available on mobile phones and wireless sensors. Traditionally, developers have addressed this problem by using low-level programming languages, such as C, to write efficient native code. Moreover, developers use low-level primitives, such as malloc/free and threads, to manage memory and concurrency. As a result, resource management becomes tedious and complex and is a source of programming errors that are difficult to identify and fix. This approach sacrifices programability and robustness in favor of performance. In this paper, we demonstrate that this trade-off is unnecessary even on a virtual machine in an optimized stream-processing (SP) environment.

The aim of the CSense is to provide developers a compiler and run-time environment that simplify the development of mHealth systems that are *robust* and support *high-rate* SP. CSense adopts a SP model similar to those proposed for flexible routers [3], multimedia applications [4], and mobile sensing applications [5]–[7]. The basic building blocks of CSense are reusable components that encapsulate user code. An application is built by connecting components into a directed acyclic graph called the *Stream Flow Graph* (SFG).

The need to simplify development and prevent bugs motivates several design decisions that distinguish CSense from prior works. We opt to implement CSense in Java – a language that provides higher productivity than C/C++ [8] – and target widely used Android devices. The key feature of the CSense model is the SFG that supports flexible application configuration, static analysis, and optimization. The CSense compiler may detect and prevent a range of programming errors including application composition errors, incorrect usage of the memory managed system, and data races. We provide a high-level mechanism for specifying concurrency by defining execution constraints among components. For example, constraints may specify the components that must be executed in different domains (i.e., threads). The compiler partitions the application into domains subject to the specified constraints and ensures that data exchanges across domains are thread-safe. High concurrency is achieved by executing multiple domains concurrently and by incorporating support for event scheduling and non-blocking I/O (NIO) operations.

Experiments show that garbage collection and concurrency mechanisms limit the scalability of SP systems on virtual machines (VMs). The memory organization of frames is important to minimize expensive copy operations and allow components to be executed at different rates. Moreover, CPU-intensive components that perform functions, such as feature extraction, typically dominate a system's workload. Such components should be implemented in native code to improve performance.

We developed a run-time environment that supports high-rate SP efficiently on the Dalvik VM. The run-time environment has the following salient features. (1) CSense uses pass-by-reference semantics and captures memory operations

explicitly as part of SFG to mitigate the impact of object creation and garbage collection. (2) We implement the run-time environment using lock-free concurrency and integrate it with Android's power management. (3) CSense optimizes the memory allocation of frames to ensure efficient frame exchanges across components. (4) CSense components may be implemented as MATLAB functions that are compiled into native code.

The performance of the CSense run-time and optimizations were evaluate empirically on mobile phones. Experiments show that the use of memory pools and lock-free synchronization improves the peak SP processing rate by as much as 25 times over a realistic baseline implementation. Moreover, our frame optimization reduces the number of memory copies and allows components to be executed at different rates. This can further improve performance by as much as 45%. More importantly, we have used CSense to implement three mHealth systems: ActiSense, AudioSense, and SpeakerSense. The three systems were selected because they produce different types of workloads and pose different system challenges. ActiSense requires high concurrency to predict patient activity from multiple accelerometers connected to a phone over Bluetooth. SpeakerSense is a CPU-intensive application that processes speech samples to determine the identity of speakers. AudioSense delivers electronic surveys and collects audio samples to evaluate the performance of hearing aids. The key challenge of AudioSense is energy efficiency, as surveys must be delivered during weeklong data collection sessions.

II. Related Work

This section reviews prior work on stream processing (SP), focusing on the differences in programming models, concurrency, memory management, and operating environment.

SP models have been studied for decades (see [9] for a review). SP systems can be broadly divided into synchronous and asynchronous systems. Synchronous systems operate on a shared clock (or clocks) that dictates when components are executed. The rigid timing of synchronous systems is suitable for compiler optimizations. Compilers can determine execution rates, buffering requirements, and implement efficient scheduling [5], [9], [10]. Asynchronous systems provide a more flexible concurrency model but may sacrifice performance, as many of the optimizations developed for synchronous systems do not translate these systems. CSense adopts an asynchronous model to support the highly concurrent workload characteristic of mHealth systems. Moreover, in contrast to some SP models that limit the structure of the data flow graph [5], [10] or develop specialized architectures [6], CSense uses general directed acyclic graphs. The push and pull semantics used to exchange frames in CSense are similar to those of Click [3]; however, pull operations are implementing as polling requests that may respond asynchronously. The SFG is the main novelty of CSense. The SFG captures application-level properties including the flow of data between components, constraints on frame types and their sizes, and concurrency. SFGs support flexible configuration, program analysis for safety, and application-level performance optimizations.

Since SP models capture parallelism explicitly they provide a rich foundation for developing highly parallel systems. The problem of partitioning streams onto threads or multiple processors has attracted significant attention [11], [12]. A majority of existing approaches require profiling information to drive the partitioning algorithms [13]. The focus of CSense is to develop a simple concurrency model and associated compiler techniques to identify likely race conditions. CSense does not use profiling information but rather utilizes constraints supplied the programmer. This is motivated by our desire to support a rapid development process, which does not allow for extensive profiling. CSense supports task parallelism by allowing for multiple execution domains. High concurrency is achieved by using events and NIO operations within a domain.

CSense employs framing optimizations to support high rate SP. The SigSeg abstract data structure used in XStream [14] and WaveScript [10] is closest to our framing optimizations. However, in contrast to SigSegs that are designed to support operations on streams at run-time, we leverage on static compiler analysis to optimize the organization of frames in memory, which allows for highly efficient implementations, albeit, at the cost of some flexibility.

Recently, a number of continuous sensing systems for mobile phones have been proposed. These systems address the challenge of running sensing applications concurrently with other phone applications [6], [7]. However, our interactions with doctors and experience with deploying mHealth systems indicate that doctors prefer to use dedicated devices in clinical trials to ensure robustness and increased battery lifetime. For this reason, CSense does not focus on resource contention between applications. More importantly, unlike most other SP systems [5], [6], [10], [14], CSense is implemented in Java to support rapid development. This introduces important system challenges that must be overcome to support high rate SP. CSense provides a 25 times higher SP rate than a baseline Java implementation by optimizing memory accesses, frame allocations, and using lock-free concurrency. CSense integrates with MATLAB to generate efficient signal processing code and with Android power locks for power management.

III. CSense Design

CSense aims to simplify the development of mHealth systems that are robust and support high-rate SP. The building blocks of CSense are fine-grained *components* that encapsulate user functionality. Applications are built by connecting components into directed acyclic graphs. This general organization is common in SP frameworks [3], [5]–[7], [14]. The novelty of our programming model is the *Stream Flow Graph* (SFG) and the associated compiler analyses and optimizations it supports.

Three basic principles underline the design of CSense:

CSense builds on Java: CSense components are implemented as Java classes. The Android SDK provides programmers a rich set of reusable components which, when used in conjunction with object-oriented programming techniques, can significantly reduce applications development time. However, this approach has disadvantages: (1) supporting high-rate SP requires careful engineering and deep understanding of the operating system internals, (2) low-level concurrency primitives

do little to support the writing of safe code, and (3) it is difficult for compilers to analyze and optimize an application globally when it is structured as loosely coupled Java components. CSense is designed to address these limitations.

Flexible, safe, and optimized applications: Applications are modeled as SFGs, which capture application-level properties including the flow of data between components, constraints on frame types and their sizes, and concurrency. SFGs support flexible configuration, program analysis for safety, and application-level performance optimizations.

Native code and profiling support: Most stream operations can be implemented efficiently in Java. However, there are cases when native implementations would significantly reduce computational overhead. CSense components may be implemented in MATLAB and compiled to native code. This has the advantage of including efficient signal processing functions that are often readily available as MATLAB toolboxes.

The remainder of this section describes the programming model and associated compiler analyses and optimizations. The run-time environment is described in Section IV.

A. Programming Model

CSense applications are built by writing and assembling *components*. Components encapsulate functionality common to mHealth systems including support for data collection, feature extraction, file I/O, and network operations. Components are connected through *connections* that are used to exchange *frames*. The run-time environment implements components as Java objects, connections as objects references, and frames are exchanged through virtual function calls.

The core abstraction of CSense is the SFG. The SFG is a directed acyclic graph (DAG) that has *components* as vertices and *connections* as edges. SFG plays a similar role to that of abstract syntax trees in traditional compilers. SFGs provide an abstraction to support application configuration, error checking, and optimization. CSense does not create yet another language for defining, configuring, and connecting components. Instead, the programmer writes a `bootstrap` program that constructs the SFG using a simple Java API. Running the `bootstrap` invokes the compiler to generate code that is then compiled for the target platform (see Section III-F). Figure 1 shows the `bootstrap` of a speaker identification system.

A *component* specification includes: the component class, port declarations, internal connections, and initialization parameters. The component class specifies the underlying Java class that implements the component. A component's port declaration defines its input and output ports, which constitute the public interface of the component. Each port has a unique name and a CSense type. A component may maintain private state. A component operates as follows: it receives a frame over an input port, transforms the incoming frame and/or updates its private state, and outputs the frame on one or more output ports. The flow of frames within a component – from one input port to one or more output ports – is captured using internal connections. This information would be difficult to extract since our components are implemented in both Java and MATLAB. Thereby, we rely on developers to specify a component's internal connections. This information is required to capture the flow of frames across components and it is used for error checking and optimizations. For the same reason, we do not allow components to copy frames internally. Initialization parameters are passed to the underlying object implementing the component when the application is initialized.

A *connection* is established from a single input port to a single output port. SFG does not permit fan-outs – a connection from a single output to multiple inputs – or fan-ins – a connection from multiple outputs to single input. Connections support bi-directional communication using "push" and "pull" semantics. A component may "push" frames to the next component over an output port; conversely, a component may "pull" data from a predecessor component over an input port. Pulls are implemented as polling requests and the components may respond asynchronously.

Components are intended to be fine-grained to maximize reuse. Additionally, CSense supports component *groups*. A group encapsulates a subgraph of connected components. The group hides its internal components and their connections and, similar to components, it exposes an interface defined by its ports. Groups constitute "syntactic sugar" and are flattened in the early stages of compilation.

B. Memory Management

The overhead of memory operations, including object creation, copy, and garbage collection, can dwarf computation times. Our experiments indicate that without proper memory management, the SP rate may be reduced by 25 times. These steep performance penalties motivate the design of our memory management system.

CSense adopts pass-by-reference semantics to exchange frames between components efficiently. For memory management purposes, we distinguish three types of components: *sources*, *user components*, and *taps*. Sources are the only components that produce new frames. Frames are modified by user components and passed to a `Tap` when they are no longer used. User components cannot make copies of frames or create new ones. We provide the `Copy` and `Ref` components to allow programmers to make copies and references of frames. We expect to raise the programmer's awareness of memory operations by forcing them to include sources, taps, `Copy`, and `Ref` in the SFG explicitly.

The explicit inclusion of memory operations as components in SFG has the advantage of allowing the compiler to analyze the application-level flow of data. At compile time, the compiler verifies that all paths that originate at a source terminate with a `Tap`. The generation of paths must account for the fact that frame references may be created either by `Ref` components or within components as captured by their internal connections. This analysis ensures that all generated frames are freed, thereby, preventing memory management bugs.

C. Type System

Our experience with developing mHealth systems has showed us that Java's type system is not sufficiently expressive

The `bootstrap` program for SpeakerSense:

```
public static void main(String[] args) {
  Project proj = new Project("mfcc.xml", "android");
  VectorC speechT = TypeC.newFloatVector(1024);
  VectorC featureT = TypeC.newFloatVector(128);

  proj.addComponent("audio",
       new AudioComponent(16000, 16));
  proj.addComponent("speechDetector",
       new SpeechDetector(16000));
  proj.addComponent("mfcc",
       new MFCCFeaturesG(speechT, featureT));
  proj.addComponent("toDisk",
      new ToDiskComponentC(featureT));
  proj.addComponent("httpPost",
      new HttpPostC("http://wsn.cs.uiowa.edu/", "fileType"));
  proj.link("audio", "energy");
  proj.toTap("speechDetector::noSpeech");
  proj.link("speechDetector::hasSpeech", "mfcc");
  proj.toTap("mfcc::out");
  proj.link("mfcc::features", "toDisk");
  proj.toTap("toDisk");
}
```

Stream Flow Graph of SpeakerSense:

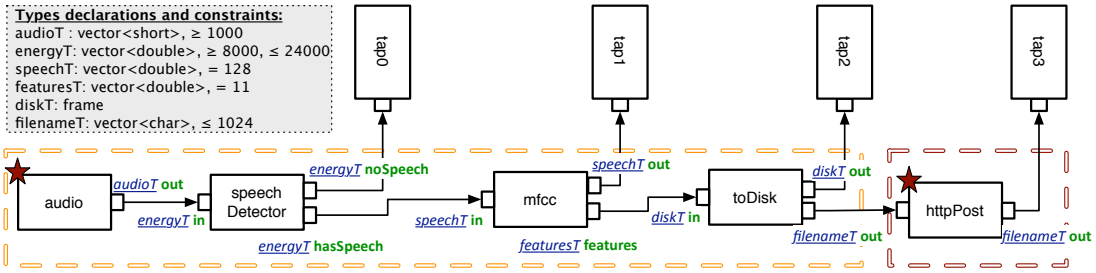


Fig. 1. The SpeakerSense `bootstrap` generates the speaker identification system. The `audio` component records audio at a configurable frequency. Sound frames unlikely to contain speech are filtered out by the `speechDetector`. The `mfcc` component computes Mel-Frequency Cepstral Coefficients. For efficiency, both `speechDetector` and mfcc are implemented in MATLAB. MFCCs are persisted on disk by `toDisk` and uploaded to a server by `httpPost` for the final speaker identification. The types of ports are denoted using underlined text and their constraints are shown in the grayed box. The `audio` and `httpPost` components require to be executed in different domains. The bounded boxes denote the execution domains.

to capture their rich framing constraints. Specifically, we found that the frame size (i.e., number of samples in a frame) is an important configuration parameter for components. Let us consider the framing constraints of the `audio`, `mfcc`, and `speechDetector` components in Figure 1. The `audio` component records sounds in an underlying frame that is returned to the user when it is full. Android OS enforces a minimum size for the recording buffer to reduce CPU utilization when recording audio. Similarly, the `speechDetector` detector can determine whether or not a frame may contain speech when its frame size is between 8,000 to 24,000 samples. Existing streaming models do not capture these constraints. As a result, configuration errors introduced during application composition may go undetected until run-time. By capturing the framing constraints explicitly, CSense can detect such misconfigurations at compile-time.

Motivate by this limitation, CSense uses an extended type system. CSense types may be divided into three categories: primitive types, array types, and Java classes. We support the same primitive types as Java: byte, short, int, long, char, float, double, and boolean. In contrast to Java's type system, we also support multi-dimensional arrays. Multi-dimension arrays are defined over primitive types and stored in column-major order. The inclusion of multi-dimension arrays simplifies integration with MATLAB. Array types are the prevalent mechanism for specifying the type of frames in CSense.

Programming languages such as C, C++, or Java do not include the size of an array as part of the type signature. CSense allows the programmer to define simple constraints ($\leq$, $<$, $=$, $>$, and $\geq$) over the size of each dimension of an array. This mechanism allows developers to write components that can be parameterized to work with frames of different size. Obviously, the size of an array type must be eventually determined. We call the procedure that determines the sizes of frame subject to the defined constraints *type materialization*. The flexibility of using size constraints should foster component reuse. A description of type materialization is postponed to next section.

Generic types allow programmers to write generic components that work with multiple types. We support generic components by allowing the developer to parameterize the type of ports. We found this mechanism to provide significant flexibility and it is used in our components extensively. For example, the `mfcc` component may be parameterized to operate either on vectors whose element type is either float or double. This allows the developer to trade-off computational accuracy of MFCCs and computational overhead. Similarly, depending on its configuration, the `audio` component may record samples as bytes (8-bit samples) or shorts (16-bit samples). This allows `audio` to be configured based on the type of its input ports.

The CSense compiler uses the type system to ensure that connections are established between compatible ports. Ports are determined to be compatible if their types are compatible. Type compatibility is determined only after type materialization when the constraints are solved. For two types to be compatible, they must fall within the same type category: primitives, arrays, or Java classes. Two primitive or Java types are compatible if they are equal, while two array types are compatible if they have the same base type and frame size.

D. Flow Analysis

Type materialization requires that the compiler determine the size of array types subject to the constraints specified by developers. However, not all feasible solutions to this

problem can be implemented efficiently. A source of inefficiency is *frame conversions* that occur when connecting ports with incompatible types. Consider the connection between the `speechDetector` and the `mfcc` components in Figure 2. A feasible solution is for the `speechDetector` to output frames of 10,000 samples and the `mfcc` to accept as input frames of 256 samples. To handle the mismatch in frame sizes, the compiler must introduce a `Converter` that receives frames of 10,000 samples and outputs frames of 256 samples. Since 10,000 is not a multiple of 256, the `Converter` cannot be implemented efficiently as it requires at least some samples to be copied. In contrast, if the `speechDetector` were to output a frame of 10,240 samples then the samples can be divided into 40 vectors that contain 256 samples as required by `mfcc`. This may be implemented without copying by defining 40 non-overlapping views over the same underlying memory buffer containing the 10,240 samples.

The goal of flow analysis is to find a solution to the type materialization problem that may be implemented efficiently. Accordingly, as part of flow analysis, the compiler determines the sizes of array types and ensures that frame conversions may be implemented efficiently. For clarity, we restrict our focus to one-dimensional arrays that have the same element type. A path captures the flow of frames in SFG from a source, through user components, and terminating with a `Tap`. To determine frame sizes and configuration parameters of frame conversions, we introduce the following variables: *super-frames* (S), *frames* (F), and *multipliers* (M) (see Figure 2). A super-frame represents a contiguous block of memory that may be divided into an integer number of frames. Since each path is analyzed independently, without loss of generality, we consider a single path that has a super-frame of size S. Four variables control the frame conversion on a connection c of the path: f_c^I, f_c^O, m_c^I, and m_c^O. The conversion is efficient when S can be divided into both m_c^I frames of size f_c^I and m_c^O frames of size f_c^O ($S = f_c^O \cdot m_c^O = f_c^I \cdot m_c^I$)

The compiler casts the problem of determining the super-frames, frames, and multipliers as an Integer Linear Program (ILP). Integer linear constraints are generated based on the type constraints supplied by the programmer according to the pseudocode shown in Figure 2. Let C_c^p be the type constraints of port p ($p \in \{I, O\}$) pertaining to connection c. As determined by our type system, a constraint is of form $(operator, v)$ where the possible operators are $(<, \leq, =, \geq, >)$ and the v is an integer. The algorithm iterates through each type constraint adding new constraints to the ILP problem. If the constraint is "$=$" (lines 4 – 7), then size of the frame (f_c^p) is set to equal the v (as specified by the type constraint) and we ensure that the super-frame (S) is a multiple of f_c^p. The value of the multiplier m_c^p will be optimized based the constraints of entire path. If the user does not supply a "=" constraint i.e., when $hasEquals = False$ (lines 14 – 16), then we set $m_c^p = 1$ indicating that the component can process the entire super-frame in a single call. In this case, the value of the frame f_c^p will be optimized based on the constraints of the entire path. If the constrain is "$>$" (lines 8 – 10), then the size of the frame (f_c^p) must exceed the user specified v. Analogous constraints are added for the case "$<$", "$\leq$", and "$\geq$". Moreover, the frames sizes f_c^p are further constraint to be smaller or equal to then super-frame sizes S (line 2). Similarly, all multipliers (m_c^p) must be at least 1. The objective function is to minimize the size of the super-frame as to minimize memory usage.

```
 1:for each constraint of C_c^p:
 2:     ILP: 0 < f_c^p ≤ S
 3:     ILP: m_c^p ≥ 1
 4:     (operator, v) = constraint
 5:     hasEquals = False
 6:     if operator is '=':
 7:          hasEquals = True
 8:          ILP:  f_c^p = v
 9:          ILP:  S = f_c^p × m_c^p
10:     elseif operator is '>':
11:          ILP:  f_c^p > v
12:     elseif operator is '<':
13:          ILP:  f_c^p < v
14:if hasEquals = False:
15:          ILP:  m_c^p = 1
16:          ILP:  S = f_c^p × m_c^p
```

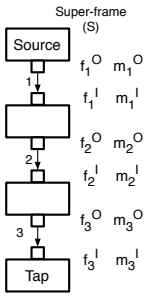


Fig. 2. Frame analysis: Algorithm and notation

Solving the created ILP will determine the value of super-frames, frames, and multipliers subject to the type constraints specified by the programmer and those required to perform efficient frame conversions. A solution to the ILP problem does not exist in two cases: there is no solution to type materialization and there is no efficient implementation. In the former case, the compiler generates an error; in the latter case the compiler generates a warning that inefficient conversions are used and reruns the ILP without the efficient frame conversion constraints. In practice, the developers select frame sizes to be multiples of each other, in which case, a feasible solution to the ILP problem exists.

E. Concurrency

Concurrency is prevalent in mHealth systems: sensors are sampled, data is uploaded to servers, and the user interacts with the underlying system. This results in a mix of events and SP operations, which must be processed concurrently.

CSense provides three concurrency mechanisms: *domains*, *events*, and *selectors*. A *domain* includes a subgraph of components that are executed in the same thread. Components pertaining to the same domain exchange frames through function calls without requiring synchronization. Data exchanges across domains are mediated by synchronization queues. Synchronization queues buffer frames to handle variations in the execution rate of different domains. A key advantage of the domain abstraction is its simplicity: the developer can reason about the behavior of components within a domain using sequential semantics.

Each domain has a *scheduler* that is responsible for managing events and selectors. Both mechanisms allow components to defer their execution to allow other components to run. CSense supports high concurrency by integrating with Java Nonblocking I/O (NIO). A component may register NIO selectors with the scheduler. The scheduler calls the component when the NIO selector has data available to read or write.

This mechanism allows the scheduler to hide most of the I/O penalties. We restrict components to be able schedule events or register selectors for themselves, essentially providing independent event streams per component. Moreover, to preserve the integrity of the domain abstraction, event and selector handlers are executed in the domain of the component. Event and selector handlers are executed without preemption.

The CSense concurrency model allows many race conditions to be detected at compile time. The model guarantees that there are no race conditions if a frame and all its references are processed within the same domain. This is because the entire subgraph of components that access the frame is embedded within a single domain. Similarly, no race conditions exist when there is a single reference to a frame, which may be accessed in one or more domains. The requirement of a single reference guarantees that the frame is processed in a single domain at any time. There exists a potential for race conditions when references to a frame are passed to different domains. In this case, the compiler issues a warning. This race analysis guarantees that when no errors are generated there are no race conditions, assuming the developer only uses the our concurrency mechanisms. However, our analysis may have false positives i.e., the compiler may issue a warning when a race does not exist. For instance, this may occur when a frame is accessed from two domains, but the two domains never execute concurrently.

The programmer can specify concurrency by defining constraints on components. First, the programmer may specify that a component should be executed in a new domain using a `NEW_DOMAIN` constraint. The constraint is associated with sources and components that include long/blocking operations. For example, the `NEW_DOMAIN` constraint may be added to the `audio` and `httpPost` to record and upload data concurrently. This is the prevalent concurrency constraint in our systems. Second, the programmer may enforce that components are executed within the same domain using a `SAME_DOMAIN` constraint. This is important in when components are tightly coupled. For example, a data source may produce samples that must be scaled. Separating this functionality into two components would foster reuse, but would introduce significant overhead at high sampling rates. In this case, the programmer can include a group containing the two components and add a `SAME_DOMAIN` constraint.

The compiler uses a simple heuristic to partition the SFG into domains subject to the specified concurrency constraints. The algorithm operates on the SFG in which all groups are flattened except for those that include a `SAME_DOMAIN` constraint. The algorithm iterates through each source in SFG assigning multiple components to a domain. Initially, the domain is set to zero and incremented in each iteration of the algorithm. Let c and d be the source and domain currently under consideration. The algorithm assigns c to run in d. Additionally, it computes the predecessor subgraph of c that includes all components x such that there is a path from x to c. If no component in the predecessor subgraph requires a `NEW_DOMAIN`, all components of the subgraph will be executed in d. Otherwise, they will be assigned to a domain in a later iteration of the algorithm. Next, the algorithm computes the successor subgraph of c that includes all components x such that there is a path from c to x. Component x will be executed in domain d if no component on the path from c to x has a `NEW_DOMAIN` constraint. In a post-processing step, the groups with `SAME_DOMAIN` constraints are flattened and the members assigned to the group's domain. The proposed heuristic typically assigns subgraphs of components that access the same frame to the same domain, which minimizes the likelihood of race conditions.

F. Compiler

The compiler has the following workflow. The `bootstrap` creates an SFG by instantiating components, and configuring and connecting them. Next, the `bootstrap` invokes the compiler. After flattening groups, the compiler checks that the SFG is structurally correct: no ports are unconnected and no fan-ins, fan-outs, or cycles exist. Additionally, we ensure that the all paths start with a source and end with a tap. The compiler runs the flow analysis to materialize types and includes `Converter` components in SFG, as appropriate. The SFG is partitioned in domains and then checked for race conditions. The final step of compilation is code generation for the target platform.

The compiler uses the MATLAB compiler to generate C code for components that use MATLAB functions. The C code is compiled as a static library. The general strategy for including a MATLAB function as a CSense component is to create a mapping between input/output ports of the component and the input arguments/return values of the MATLAB function. This is accomplished through the same API used to configure SFGs. The compiler generates custom wrapper classes that call the generated static library. Data is exchanged using NIO buffers for efficiency. The compiler also generates a "main" application that configures components, connects them, and creates threads for their execution. The code generation completes by compiling the generated code. Even though in this paper we focus on Android, owing to Java's portability, we have been able to run CSense on both Linux and OS X.

IV. Run-time Environment

A. Component Implementation

CSense components subclasses the Java class `CSenseComponent`, which provides about 20 functions that cover life-cycle management, frame manipulation, event handling, and logging. We provide convenient default implementations for all these functions. Typically, components override `onPush`, `onPull`, and `onEvent`. The `onPush` and `onPull` handlers are called when frames are pushed or pulled. The `onEvent` handler delivers events that were previously scheduled by the component.

CSense components exchange the majority of data over connections. Occasionally, components benefit from exporting other public methods. For example, the audio component may expose methods to control when sound is recorded. Other components may access this interface by obtaining a reference to an instance of an audio component by name. Additionally, components may communicate through publish/subscribe

channels. This mechanism intended for components to publish data to be shared with multiple subscriber components.

B. Scheduler

An application is partitioned into domains, each domain having its own scheduler. The scheduler is responsible for managing memory, events, and selectors.

The goal of memory management is to minimize the impact of object creation, copying, and garbage collection. We implement memory management as follow. Each source maintains a memory pool that contains a number of super-frames, which are preallocated when the application starts. A source retrieves a super-frame from the pool when it has data to write. Flow analysis ensures that frames are exchanged efficiently until they reach a tap. Upon reaching the tap, the scheduler must determine if it should put the super-frame back in the memory pool. We associate a reference counter with each super-frame. The reference counter is incremented each time a new reference is created. Conversely, the counter is decremented when taps are reached and, when the counter becomes zero, the super-frame is put back in the pool for reuse. This mechanism limits the creation of new frames and their garbage collection.

A component may schedule events to run after a delay. The scheduler maintains two execution queues. The immediate execution queue is a FIFO queue that stores zero delay events. Components use zero delay events to yield their turn and allow other components to be executed. Non-zero delay events are inserted in a priority queue sorted by time when they are scheduled to fire. The scheduler operates in rounds. In each round, the scheduler drains the immediate queue and processes all the events in the priority queue scheduled to execute before the current time. A component may also register selectors with the scheduler. Selectors are checked at the end of a round and components that have pending data are notified.

Memory pools and events may be accessed from different threads, so a concurrency mechanism is necessary. Java includes support for concurrent collections including blocking queues and synchronized arrays that may be used to implement the event queues and memory pools of the scheduler. However, the underlying implementation of these data structures use reentrant locks. Locks are designed to handle high levels of contention. Under low or medium contention, locks introduce a high overhead since a thread must be suspended when it attempts to acquire a lock that is already held by a different thread. Atomic variables provide a lightweight synchronization mechanism that is implemented efficiently using hardware supported compare-and-swap. The challenge with atomic variables is that the developer has to implement appropriate mechanism to handle concurrent access. To improve SP rates, we have implemented customized synchronization primitives. Our synchronization primitives use a two-level locking scheme. Atomic variables are used for concurrency in the low content case. If the lock implemented using atomic variables is not acquired after several attempts, we switch to using reentrant locks. A comparison between the concurrent collections provided by Java and our customized primitives is included in Section V-A

C. Android Integration

CSense is designed to take advantage of the underlying Android services. Consistent with the Android architecture, a CSense application uses activities for user interfaces and a service to host its run-time environment. The user interface and service run in the same process, but in different threads. CSense components have specialized implementations for Android. For example, components that use sensors leverage on the Android APIs to capture motion, GPS, and audio data.

CSense integrates with Android's power management to allow phones to sleep. Android uses power locks to prevent the CPU and display from entering a sleep state. When no power locks are acquired, Android will aggressively turn them off. Releasing power locks prematurely may result in an application being suspended for an indeterminate amount of time. Other resources, such as network or GPS are not managed though power locks. Instead, the programmer must explicitly turn them on and off. These resources are typically accessed from a single CSense component that is also responsible for managing their power.

There are two challenges to integrating our scheduler and Android's power management: (1) we must determine when it is safe to sleep, and (2) we must develop an efficient mechanism to enter and leave sleep states. To determine if it is safe to sleep, the scheduler consults the pending events and registered I/O handlers. Each scheduler maintains an independent power lock that is acquired during its initialization. In the following, we describe the behavior of each scheduler independently. The CPU will sleep only when *all* schedulers release their power locks. Let t_{now} be the current system time and t_{first} be the time when the next event in either one of the scheduler's queues is scheduled to run. If $t_{now} \leq t_{first}$, then the scheduler is running behind, effectively having to catch up with the sequence of events. Thus, the power lock cannot be released to allow the scheduler to catch up. Otherwise, if $t_{now} > t_{first}$, the scheduler can sleep for $d = t_{first} - t_{now}$ seconds. In this case, the scheduler registers an alarm to wake up the system after d seconds. Android guarantees that alarms wakeup the system from sleep, at which point, the scheduler reacquires the power lock. In the case when no events are scheduled, the scheduler will go to sleep and may be woken by receiving data from other domains or by external events.

Initial testing indicated that the above algorithm has important limitation: it does not account for the time necessary to transition to sleep and then to wakeup. Let t_{wakeup} be the time from the time when the power lock is released until the wakeup alarm is delivered. If the time the scheduler may sleep $d < t_{wakeup}$, then some events will be delivered late. This is particularly problematic when there are numerous events to be processed due to highly concurrent workloads. To address this limitation, we devised a two-level sleep strategy that only release the power lock when $d > t_{th}$, where t_{th} is user-specified constant. If $d < t_{th}$, then the scheduler will use Java's wait/notify mechanism to sleep for d seconds without releasing the power locks. Otherwise, we release the power locks and allow the CPU to sleep. This algorithm is safe in that it does not introduce additional delay penalties to pending

events due to sleep.

V. Evaluation

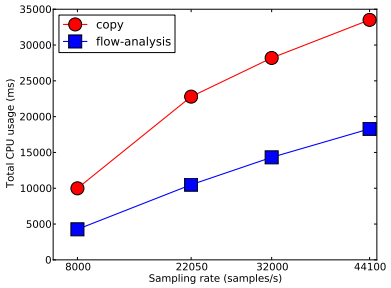
In this section, we provide an empirical evaluation on Galaxy Nexus phones running Android Jelly Bean. Galaxy Nexus use a Texas Instruments OMAP 4460 SoC that includes a 1.2 Ghz dual-core ARM Context-A9. The phone has 1GB of memory and 32 GB of storage. C code is generated from MATLAB functions using MATLAB R2012b and MATLAB Coder 2.3. The resulting code is cross-compiled into a static library using Android NDK (r8d). Microbenchmarks are designed to highlight the importance of memory management, concurrency mechanisms, and efficient frame conversions. Additionally, we used CSense to develop three complete systems to evaluate its ease of use and efficiency.

A. Microbenchmarks

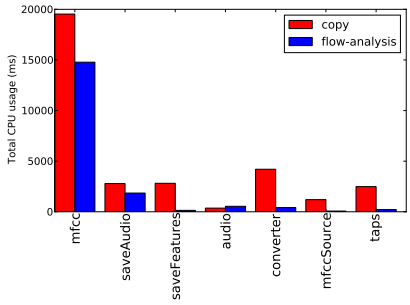
Producer-Consumer Benchmark: We implemented a Producer-Consumer benchmark to evaluate the performance of the CSense scheduler. The producer generates frames at specified rates. The produced frames are passed to the consumer and then to a tap. The producer and consumer operate in different domains to capture the impact of inter-domain connections. Memory was managed using either Java's memory management (GC) or using memory pools (MP). In the former case, new objects are created for each frame and we rely on garbage collection to free them. This benchmark does not include any frame conversions. Concurrency in the scheduler was implemented using locking primitives (L) and CSense's synchronization primitives (C). A scheduler implementation combines a memory management and a locking mechanism. Implementations are labeled accordingly. The reported results are averages over 1 minute traces.

Figure 3(a) shows the performance of three schedulers. We increase the offered rate linearly and measure the rate at which the consumer receives events. A scheduler should match the offered rate until it reaches its peak rate. To understand the source of the performance differences between implementations, we also measure the total garbage collection time and CPU usage. The CPU usage is measured as the total time the benchmark runs on either CPU core.

A naive implementation of the scheduler would use Java's memory management and locking concurrency primitives (`GC+L`). Unfortunately, `GC+L` performs poorly: it supports a peak rate of only 1,336 events/s. `MP+L` incorporates memory pools but continues to rely on locking concurrency primitives. Memory pools eliminate the creation of frames and reduce garbage collection. This approach supports a peak event rate 20,518 events/s, which is 15 times higher than the naive implementation. Figure 3(b) plots the garbage collection time for each implementation as reported by Dalvik VM. As expected, the naive implementation has the highest garbage collection time. `MP+L` has significantly lower garbage collection time, but garbage collection is not eliminated. In fact, the garbage collection time increases linearly with the offered rates, albeit at a slow rate. The source of the garbage collected objects is the `ReentrantLock` used in Java concurrent collections.



(a) Benefits of flow analysis



(b) Detailed performance at 44100 Hz

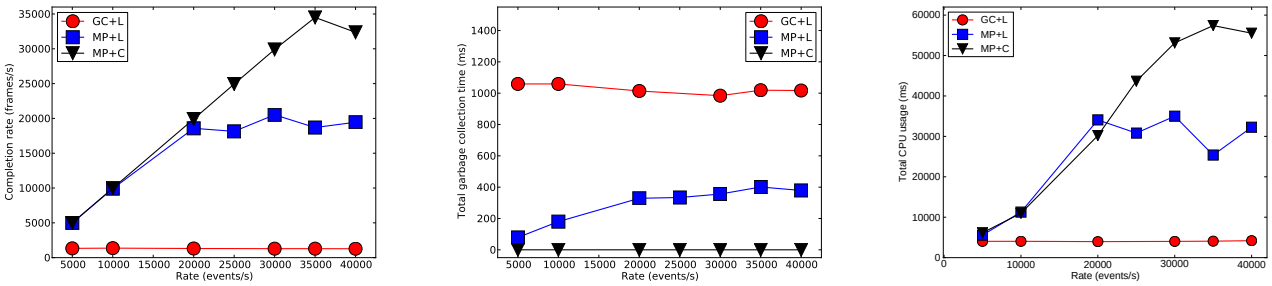
Fig. 4. MFCC benchmark: Impact of whole-system optimizations.

These objects are created when a thread attempts to access a lock that is already held by a different thread. This justifies the linear increase in garbage collection times observed when the offered rate is lower than the peak rate.

Using our concurrency primitives, the scheduler (`MP+C`) may support a peak rate of 34,503 events/s, which represents an additional 40% improvement over `MP+L`. Overall, the proposed optimizations provide a 25 times improvement over the naive implementation. Two factors contribute to these improvements. The garbage collection time is reduced to zero when our customized synchronization primitives are used. Additionally, as shown in Figure 3(c), `MP+C` runs for a longer time as indicated by the higher CPU time. This is because our synchronization primitives reduce the number of thread suspensions and resumptions.

Result: *Object creation, garbage collection, and locking concurrency primitives limit the scalability of SP on VMs. Memory pools and lock-free concurrency may be used to increase the supported peak event rate by as much as 25 times.*

MFCC microbenchmark: To evaluate the benefits of flow analysis, we implemented a pipeline that computes MFCCs from audio samples and saves them to disk. The pipeline is similar to that shown in Figure 1 except that we omit the `speechDetector` to ensure a consistent workload in all experiments. This functionality is representative of the feature extraction commonly performed in mHealth applications. Moreover, the `mfcc` component is implemented in MATLAB showcasing the ability to include MATLAB code in CSense applications. We configured the run-time environment to use memory pools and our customized concurrency primitives. We provide results for when frame conversions are implemented using memory copies and our frame analysis.

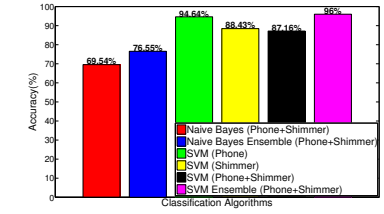


(a) Scheduler implementations (b) Total garbage collection time (c) CPU usage

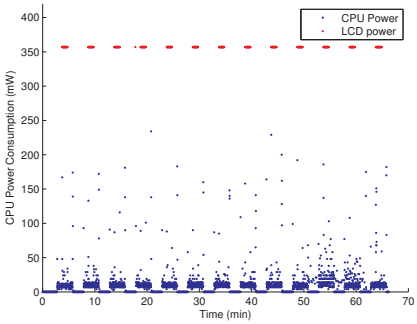
Fig. 3. Producer-consumer benchmark: Assessing the impact memory pools and concurrency mechanisms.

Figure 4(a) plots the CPU usage when the audio sampling rate was 8000, 22050, 32000, and 44100 Hz. The figure clearly indicates the benefits of using the efficient frame conversions enabled by flow analysis. Moreover, these benefits increase with the audio sampling rate. At 44100 Hz, using flow analysis, the CPU usage is reduced by 45% compared to the baseline. To better understand the benefits of framing, Figure 4(b) plots the time spent in each component of the MFCC pipeline. Aside from minimizing the number of object copies, the use of super-frames has three additional advantages: (1) It reduces overhead since super-frames contain more samples than frames but require the same number of function calls to push. This results in lower overhead on `mfccSource` and `tap` components that are responsible for memory management. (2) Superframes allow components to execute at different rates. The superframe is 4096 samples, but the `mfcc` source and `saveFeatures` are executed 32 and 1 time, respectively, to process a super frame. This feature explains the reductions in the CPU time of `mfcc`, `saveFeature`, and `saveAudio`. (3) Finally, the `Converter` component is used to convert short integers to doubles. For efficiency, this component is implemented in native code.

Results: *Flow analysis permits efficient frame conversions that not only reduce the number of memory copies but also allow components to be executed at different rates.*



(a) ActiSense: Prediction activities with phone and Shimmer motes



(b) AudioSense: Power management for week-long deployments

Fig. 5. Empirical results from CSense applications

B. Macrobenmarks

We have implemented three mHealth systems using CSense: AudioSense, ActiSense, and SpeakerSense. Our discussion highlights the wide-range of features supported by CSense.

ActiSense: ActiSense is a system that infers patient activities (running, sitting, walking, standing, and climbing stairs) using accelerometer readings from multiple sensors. The system is organized as a Body Sensor Network (BSN) in which a mobile phone acts as a coordinator. Four Shimmer motes are placed on the patient's limbs. The Shimmer motes collect acceleration readings at 50 Hz and relay the raw readings to the mobile phone over Bluetooth. The mobile phone also collects acceleration readings at 60 Hz. In addition, the mobile phone is responsible for extracting features and using these features to classify the patient's activities in real-time. The computed features are mean, time-domain and frequency-domain entropy, and correlation features. Feature extraction components have been implemented in MATLAB and compiled to native code. A Support Vector Machine (SVM) classifier is used to predict patient activities in real-time. The main systems challenge in ActiSense is to support high concurrency to allow for data to be collected over Bluetooth and classify the patient's activity. ActiSense is based on the system developed by Bao and Intelli [15]. A working prototyped of ActiSense that collected data only from the mobile phone was developed in a single day. Figure 5(a) shows prediction accuracy of different classifiers. An accuracy of 96% is achieved when using SVM ensambles that combine both data from phones and Shimmer motes. This shows that CSense can be used to build accurate activity recognition systems.

AudioSense: AudioSense uses mobile phones to measure the performance of hearing aids in real-time and in-situ

[16]. The performance of hearing aids is characterized using electronic surveys and sensor data. At randomized intervals, electronic surveys are delivered to patients to assess their subjective assessment of the hearing aid's performance. While the patient completes a survey, AudioSense also captures audio samples and GPS locations to characterize the patient's listening context. AudioSense records audio samples at 16 Khz and GPS location 0.1 Hz. AudioSense performs some signal processing onboard and uploads the collected data over a 3G connection to a server for archival and further analysis. The workload introduced by AudioSense alternates between periods of activity and inactivity: on average data is collected for 10 minutes every 1.5 hours. Therefore, a key challenge is for CSense to closely integrate with the phone's power management system to allow data to be collected during weeklong session with minimum number of recharge cycles. Figure 5(b) shows the power consumption during which the phone delivers surveys every 5 minutes and collects data for 3 minutes. Long periods of sleep may be achieved due to the tight integration of the CSense scheduler with power locks. AudioSense is currently deployed in a clinical trial to evaluate hearing aids that will eventually include 50 patients. The current version of AudioSense can deliver surveys for 3 days without recharging batteries. Preliminary data indicates that the system is highly robust, uploading all collected data to our web server in spite of network disconnections and imperfect cellular coverage.

SpeakerSense: SpeakerSense determines the identity of speakers involved in a conversation in real-time from audio samples. Modern speaker identification systems use Gaussian Mixture Models (GMMs) to model speech, consistent with the view that speech may be classified in a small number of acoustic classes. SpeakerSense collects 16-bit samples at 16 Khz, which are used to compute the Mel-Frequency Component Coefficients (MFCC). As MFCCs are computationally intensive, SpeakerSense computes features on the mobile phone which are then uploaded to a server for further analysis. The key challenge in developing SpeakerSense is to support the efficient computation of MFCC. Moreover, integration with MATLAB on the server side simplifies classification as MATLAB has support for GMMs.

VI. Conclusions

In this paper we presented the design, implementation, and evaluation of CSense – a compiler and run-time environment that simplifies the development of mHealth systems that are *robust* and support *high-rate* SP. CSense is implemented in Java and integrates with MATLAB to simplify development. The primary contribution of this paper is a new programming model built on the Stream Flow Graph. The SFG captures application-level properties including the flow of data across components, constraints on frame types and their sizes, and concurrency. SFGs support flexible configuration, program analysis for safety, and application-level performance optimizations. The CSense compiler may detect composition errors, incorrect memory management, and data races.

We have identified that the memory management, concurrency, and power management limit the scalability of SP systems. We incorporate memory pools, frame conversion optimizations, custom synchronization primitives, and careful integration with power locks to develop a scalable run-time environment.

Microbenchmarks on Galaxy Nexus phones demonstrate the benefits of the proposed programming model, compiler optimizations, and run-time environment. Empirical results indicate that our run-time environment achieves 25 times higher SP rate compared to a baseline that uses Java's memory management and locking concurrency. Moreover, our frame conversion technique can improve performance for up to 45% in a realistic application. We demonstrated the versatility of CSense by developing three mHealth systems.

References

[1] O. Chipara, C. Lu, T. C. Bailey, and G.-C. Roman, "Reliable clinical monitoring using wireless sensor networks: experiences in a step-down hospital unit," in *SenSys*, 2010.

[2] S. Consolvo, D. W. McDonald, T. Toscos, M. Y. Chen, J. Froehlich, B. Harrison, P. Klasnja, A. LaMarca, L. LeGrand, and R. Libby, "Activity sensing in the wild: a field trial of ubifit garden," in *SIGCHI*, 2008.

[3] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[4] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *MobiSys*, 2011.

[5] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed., vol. LNCS 2304, 2002, pp. 179–196.

[6] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell, "The Jigsaw Continuous Sensing Engine for Mobile Phone Applications," in *SenSys*, 2010.

[7] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song, "SymPhoney: A Coordinated Sensing Flow Execution Engine for Concurrent Mobile Sensing Applications," in *Sensys*, 2012.

[8] P. Tyma, "Why are we using java again?" *Communications of ACM*, vol. 41, no. 6, pp. 38–42, Jun. 1998.

[9] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[10] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett, "Design and evaluation of a compiler for embedded stream programs," *ACM Sigplan Notices*, vol. 43, no. 7, p. 131, 2008.

[11] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet, "A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler," in *LCTES*, 2012, pp. 51–60.

[12] P. Hudak and A. Bloss, "The aggregate update problem in functional programming systems," in *POPL '85*, 1985, pp. 300–314.

[13] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: profile-based partitioning for sensornet applications," in *NSDI*, Apr. 2009.

[14] L. Girod, Y. Mei, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "XStream: a Signal-Oriented Data Stream Management System," in *ICDE*, 2008.

[15] L. Bao and S. Intille, "Activity recognition from user-annotated acceleration data," in *Pervasive Computing*, ser. Lecture Notes in Computer Science, 2004, vol. 3001, pp. 1–17.

[16] S. S. Hasan, F. Lai, O. Chipara, and Y.-H. Wu, "Audiosense: Enabling real-time evaluation of hearing aid technology in situ," in *CBMS*, 2013.