CSense: A Stream-Processing Toolkit for Robust and High-Rate Mobile Sensing Applications

Farley Lai, Syed Shabih Hasan, Austin Laugesen, Octav Chipara
Department of Computer Science/Aging Mind and Brain Initiative
University of Iowa
Iowa City, US
{poyuan-lai, syedshabih-hasan, austin-laugesen, octav-chipara}@uiowa.edu

***Abstract*—This paper presents CSense – a stream-processing toolkit for developing robust and high-rate mobile sensing application 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. Our compiler includes a novel flow analysis that optimizes the exchange of data across components from an application-wide perspective. A mobile sensing application benchmark indicates that flow analysis may reduce CPU utilization by as much as 45%. Static analysis is used to 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 19 times higher steam processing rate compared to a realistic baseline implementation. We demonstrate the versatility of CSense by developing three mobile sensing applications.**

***Keywords*—Dataflow computing; runtime environment; embedded software**

I. Introduction

Mobile phones are capable sensing platforms that include multi-modal sensors, increasing computational and memory resources, and versatile networking capabilities. Their capabilities have enabled a new generation of mobile sensing applications (MSAs). We are interested in using mobile phones to transform how healthcare professionals collect information regarding a patient's physiology, physical activities, and social interactions. Results of recent studies on mobile health systems have shown the feasibility of collecting medical records with higher resolution than is possible through manual data collection methods [1]–[4]. However, experience has also shown us that the development of MSAs is particularly time demanding and challenging as significant time is spent on ensuring that the system operates robustly within the resource constraints of the embedded platform. The development of MSAs faces several challenges that are poorly addressed by existing operating systems like Android:

Concurrency: MSAs must handle data processing and asynchronous events concurrently: sensors are sampled, data is uploaded to servers, and the system responds to user interactions or changes in the environment. Such systems are difficult to implement correctly using low-level concurrency primitives such as threads or events. Thus, MSAs require a flexible concurrency model that supports static analysis to detect bugs.

High Frame Rates: MSAs collect data from one or more sensors at high rates (e.g., 44100Hz for processing audio). The collected data frames must be processed in real-time or within a few seconds from collection and may involve expensive signal processing operations (e.g., FFT). Supporting such high rates is difficult due to the limited resources available on mobile phones.

Reliability: MSAs are intended for long-term data collection from users in unpredictable environments. This operating regime, coupled with the need to provide a positive user experience, motivates a focus on bug prevention to reduce runtime errors.

Java Run-time Environment: Although Java increases programmer productivity and reduces programming errors (compared to C/C++), it also increases the complexity of implementing MSA efficiently. Efficient implementations must manually manage memory, select appropriate concurrency mechanisms, and integrate native implementations of expensive operations.

To address these challenges, we propose CSense, a stream processing (SP) toolkit for developing MSAs that provides a programmer the following capabilities:

- Consistent with SP approaches [5]–[9], an MSA is modeled as a directed acyclic graph of components that encapsulate reusable user code. The programming model includes three novel features: (1) a simple but expressive concurrency model that handles concurrent operations and asynchronous events efficiently, (2) a flexible type system used to specify the types of inputs and outputs for components, and (3) the explicit inclusion of memory management operations as part of the component graph.
- Our compiler includes a flow analysis that leverages type information and explicit memory management to perform application-wide optimizations. The same information is leveraged by our static analysis to identify a range of programming errors including application composition errors, incorrect usage of the memory management system, and data races. Code generation techniques are used to integrate MATLAB functions (compiled into C code) as CSense components.
- We designed a run-time environment that supports high-rate SP on the Dalvik VM. This required careful

design of memory management and concurrency. The run-time environment tightly integrates with Android's power management system.

The primary novelty of CSense is the inclusion of type and memory management information as part of the programming model to facilitate static program analysis and optimizations.

Several frameworks aim to simplify various aspects of MSA development. CSense is closely related to efforts aimed at reducing the burden of resource management [8]–[11] and complementary to those that focus on integration with cloud services [12] or machine learning support [13], [14]. SeeMon [10] selects an informative set of sensors to track a user's context. Coordinator [11] extends these capabilities to adapt application behavior in response to resource availability. JigSaw [8] provides customized pipelines for accelerometer, microphone, and GPS sensors. A limitation of these systems is that they only support MSAs that may be defined using constrained queries [10], [11] or customized pipelines [8]. In contrast, CSense provides a high-level stream programming abstraction suitable for a broad range of MSAs. Similar to CSense, SymPhony [9] provides a general programming model for MSAs, however, it emphasizes the problem of sharing resources across multiple MSAs that run on a single device. We do not focus on resource sharing as we are interested in mobile health applications where devices are dedicated to run a single application. More importantly, SymPhony does not support flexible concurrency, compiler optimizations, or static analyses.

The benefits of the CSense run-time and optimizations have been evaluated on mobile phones. Experiments show that the use of memory pools and lock-free synchronization improves the peak SP rate by as much as 19 times over a realistic baseline implementation. Moreover, our frame analysis reduces the number of memory copies and allows components to be executed at different rates. We show that flow analysis can reduce CPU usage by as much as 45% in a realistic application.

We have used CSense to implement three MSAs: SpeakerIdentifier, ActiSense, and AudioSense. The three systems were selected because they produce different types of workloads and pose different system challenges. SpeakerIdentifier is a CPU-intensive application that processes speech samples to determine the identity of speakers. ActiSense requires high concurrency to predict patient activities from multiple accelerometers connected to a phone over Bluetooth. AudioSense [4] delivers electronic surveys and collects audio samples to evaluate the performance of hearing aids. The key challenge of AudioSense is to collect sensor data reliably during weeklong data collection sessions.

The remainder of the paper is organized as follows. The programming model, compiler analysis, and optimizations are presented in the next section. The run-time environment that executes CSense applications is described in Section III. Micro- and macro-benchmarks that show the benefits of flow optimization and run-time environment are provided in Section IV. The related work is reviewed in Section V. Conclusions are included in Section VI.

II. CSense Design

CSense supports the development of MSAs that are robust and require high-rate SP. The building blocks of CSense are fine-grained components that encapsulate user functionality. An application is built by connecting components to form *Stream Flow Graph* (SFG). The SFG includes type and memory management information that facilitate static compiler analyses and optimizations.

The design of CSense is based on the following principles:
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 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 provide little support for 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 addresses 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: 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 next section.

A. Programming Model

Components are the building block of CSense applications. They encapsulate functionality common to MSAs including support for data collection, feature extraction, file I/O, and networking operations. Applications are written by connecting components into a directed acyclic graph called the *Stream Flow Graph* (SFG). We distinguish two types of components: *modules* and *configurations*. Modules provide the underlying Java implementation of a component. Existing Java libraries may be reused as part of module implementations. Configurations may be used to either (1) configure a single module (called simple configurations) or to (2) connect and configure groups of modules and configurations to create reusable components (called group configurations). A module must have at least one configuration to be used in an application. Each application has a main group configuration that connects and configures all the modules of an application. We opted to implement both modules and configurations in Java. Using Java has many advantages including programmer familiarity, ease of integration with Android, and availability of compiler and analysis tools.

```
public class RMSClassifierM<T extends Vector>
   extends Module {
   InputPort<T> in = newInputPort(this, "in");
   OutputPort<T> above = newOutputPort(this, "above");
   OutputPort<T> below = newOutputPort(this, "below");
   double threshold;

   public RMSClassifierM(double threshold) {
     this.threshold = threshold;
   }

   public void onInput() {
     T v = in.getFrame();
     double rms = computeRMS(v);
     if (rms ≥ threshold) above.push(v);
     else below.push(v);
   }
};
```

Fig. 1. The `RMSClassifierM` module

```
1: public class RMSClassifierC
      extends SimpleConfiguration {
2:    public RMSClassifierC(double threshold) {
3:      // === specify Java implementation ===
4:      super(RMSClassifierM.class);

5:      // === type definitions ===
6:      VectorC type = TypeC.newFloatVector()
7:      type.addConstraint(Constraint.GT(8000));
8:      type.addConstraint(Constraint.LT(24000));

9:      // === ports definitions ===
10:     InputPortC in = addInputPort(type, "in");
11:     OutputPortC above = addOutputPort(type, "above");
12:     OutputPortC below = addOutputPort(type, "below");

13:     // === specify internal links ===
14:     link(in, above); link(in, below)

15:     // === add component arguments ===
16:     addArgument(new Argument(threshold));
17:   }
18:};
```

Fig. 2. Configuration of `RMSClassifierM` shown in Figure 1

An example of a module is shown in Figure 1. The `RMSClassifierM` classifies frames based on their root means square (rms) value. The core of the module is the `onInput()` function that is called when there are frames to be processed on all the module's input ports. Within the `onInput` function, the `RMSClassifierM` retrieves a frame from the `in` port and, depending on the computed rms value, the frame is pushed on either the `above` or `below` port. More complicated components may maintain private state and schedule/handle events. CSense, also supports "pull" semantics: a component may request data from upstream components by calling `pull()` on any of its input ports. Pulls are implemented as polling requests and the upstream components may respond asynchronously by scheduling events. We will return to the details of event handling in the context of the concurrency model later in this section.

A simple configuration defines the ports, internal connections, and initialization parameters of the module it configures. An example of a simple configuration is shown in Figure 2. The public interface of a component is defined by its input and output ports (lines 10 – 12). The types of ports are specified according to the type system described in Section II-D. A typical component execution involves retrieving frames from input ports, modifying these frames, and pushing them over output ports. The flow of frames *within* a component – from one input port to one or more output ports – is captured by its internal connections (line 14). A connection indicates the *potential* of a frame exchange rather than a requirement. Accordingly, a component that connects an input port (I) to two output ports (O_1 and O_2) may, at run-time, output a frame on O_1, O_2, or on both O_1 and O_2. For example, the configuration `RMSClassifierC` connects the input port `in` to both `above` and `below` ports. At run-time, the module `RMSClassifierM` will output a frame on either `above` or `below` depending on the rms value of the frame. We prohibit components from creating new frames as precise information regarding the flow of frames is necessary to perform optimizations and error checking.

Figure 3 provides an example of a group configuration that specifies a speaker identification application. The group configuration allows for components to be instantiated, configured (lines 5 – 10), and linked (lines 11–17). While groups constitute "syntactic sugar", they facilitate code reuse. For example, `MFCCFeaturesG` is a group that includes components that implement several signal processing operations. Internally, the group automatically configures the filter bank required to compute the MFCC based on the size of the feature type. These details are hidden from external components.

B. Memory Management

The overhead of memory operations, including object creation, copy, and garbage collection, can dwarf computation times as shown in Section IV. As a consequence, CSense adopts pass-by-reference semantics and incorporates memory management operations into the SFG. 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. As previously mentioned, user components are prohibited from creating or copying frames. These operations are supported by including `Copy` and `Ref` components in SFGs. We expect to raise the programmer's awareness of memory operations by incorporating explicit stream operators. More importantly, this approach allows the entire flow of frames in an application to be known at compile time allowing for optimizations and static analysis.

C. Concurrency

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

CSense provides four concurrency mechanisms: *domains*, *events*, *selectors*, and a global *workspace*. 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.

```
1: public class SpeakerIdentifierG extends GroupConfiguration {
2:  public SpeakerIdentifierG(int rateInHz,
                              URL server, double rms) {

3:   // === type definitions ===
3:   VectorC speechT = TypeC.newFloatVector(1024);
4:   VectorC featureT = TypeC.newFloatVector(128);

5:   // === add components to group ===
6:   addComponent("audio", new AudioComponentC(rateInHz, 16));
7:   addComponent("rmsClassifier", new RMSClassifierC(rms));
8:   addComponent("mfcc", new MFCCFeaturesG(speechT, featureT));
9:   addComponent("toDisk", new ToDiskComponentC(featureT));
10:  addComponent("httpPost", new HttpPostC(server, "fileType"));
11:  // === connect components ===
12:  link("audio", "rmsClassifier");
13:  toTap("rmsClassifier::below");
14:  link("rmsClassifier::above", "mfcc::sin");
15:  fromMemory("mfcc::fin");
16:  toTap("mfcc::sout");
17:  link("mfcc::fout", "toDisk");
18:  toTap("toDisk");

19:  // === specify concurrency constraints ===
20:  getComponent("audio").setThreading(Threading.NEW_DOMAIN);
21:  getComponent("httpPost").setThreading(Threading.NEW_DOMAIN);
22:  getComponent("mfcc").setThreading(Threading.SAME_DOMAIN);
23: }
```

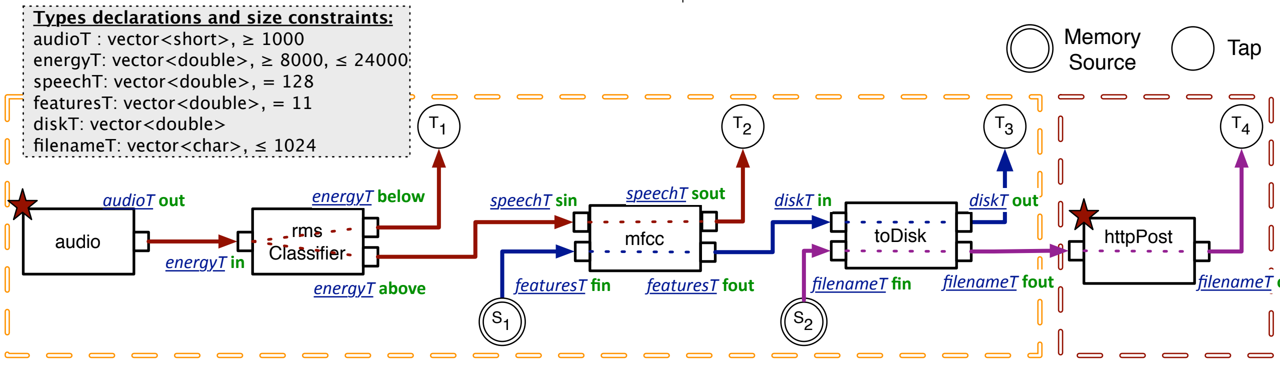

Types declarations and size constraints:
audioT : vector<short>, ≥ 1000
energyT: vector<double>, ≥ 8000, ≤ 24000
speechT: vector<double>, = 128
featuresT: vector<double>, = 11
diskT: vector<double>
filenameT: vector<char>, ≤ 1024

Fig. 3. The main configuration of a speaker identification system adapted from [15]. The `audio` component records audio at a configurable frequency. The `RMSClassifierC` filters out sound frames unlikely to contain speech. The `mfcc` component computes Mel-Frequency Cepstral Coefficients. For efficiency, `mfcc` is 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.

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 New I/O (NIO). A component may register NIO selectors with the scheduler. The scheduler calls the component when the selector has data available to read or write. This mechanism allows the scheduler to multiplex I/O requests. We restrict components to schedule events or register selectors for themselves, i.e., providing independent event streams per component. Moreover, to preserve the integrity of the domain abstraction, events and selector handlers are executed in the domain of the component that scheduled their execution. Components may share state through a global *workspace*. The workspace is organized as a dictionary in which shared variables are read and written through using agreed-upon keys.

CSense applications are multithreaded and may include shared state. As a consequence, there is a potential for race conditions to occur when variables are accessed from multiple domains. The race analysis ensures that individual accesses to shared variables occur in synchronized blocks. This invariant ensures that individual accesses to shared variables are race-free. While this approach avoids a majority of data races, there still is the potential for data races when implementing more complex synchronization protocols that involve multiple accesses to shared variables. Two factors make enforcing the above invariant in Java programs difficult: (1) determining the target of each update and (2) determining the complete set of potential execution inter-leavings. Our race analysis takes advantage of the restricted semantics of the CSense programming model to address these challenges. First, the problem of identifying the target of an update is straightforward since the programming model requires shared variables to be accessed via the workspace. Second, unlike for general Java programs whose call graph is not fully known at compile time, the call graph in CSense is encoded by the SFG. The race analysis checks that all execution paths that access a shared variable are from within synchronized blocks. This is accomplished by accounting for the fact that the only entry points for a component are the `onInput` (called to exchange frames) and `onEvent` (called to handle events) method calls. Our analysis may also 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 (lines 20 – 22 in Figure 3). Second, the programmer may enforce that components are executed within the same domain using a `SAME_DOMAIN` constraint. For example, the components implementing the `mfcc` component should not be split across domains. Otherwise, a large number of frames would have to be exchanged via synchronization queues adding significant overhead.

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 the 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 share a path to the same domain, which reduces overhead.

D. Type System

Our type system is designed to provide the programmer with flexibility in specifying frame types. A frame can be either a vector or a multi-dimensional matrix of primitive Java types. While Java does not support matrices as types, CSense supports them to simplify the integration with MATLAB. The main extension to the type system is that we allow the programmer to specify simple constraints ($\leq$, $<$, $=$, $>$, and $\geq$) over the size of each dimension of an array. These constraints may be added cleanly as part of configurations (see lines 5 – 8 in Figure 2 for an example). Obviously, the size of a frame must be eventually determined. We define *type materialization* to be the procedure that determines the frame sizes subject to the defined constraints.

The support for parameterized and constrained frame types benefits error checking, component reuse, and optimization. Let us consider the audio and mfcc components. The audio component records sound in an underlying frame that is returned to the user when it is full. The Android OS enforces a minimum size for the recording buffer to reduce CPU utilization when recording audio. In contrast, the input mfcc component outputs features that always contain 11 floats. CSense identifies configuration errors due to connections between ports of incompatible rates at compile time. In other SP models, such errors would go unnoticed until run-time.

Rich types also foster component reuse. For example, the mfcc component may be parameterized to use vectors whose element type is either float or double. This allows the developer to trade-off computational accuracy of MFCCs and computational overhead with a minor change to the frame type. 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 output ports.

Defining components that operate over ranges of sample sizes is a powerful construct. For example, audio may use frames whose size exceeds 1024. Therefore, the same audio component may be used in two applications with different size configurations. This makes it feasible to select frame sizes such that components operate optimally from an application-wide perspective. CSense may accomplish this through the flow analysis presented next.

E. Flow Analysis

Type materialization requires that the compiler determine the size of frames 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 can occur when ports, which require frames of different sizes, are connected. Consider the connection between rmsClassifier and mfcc in Figure 3. A feasible solution is for the rmsClassifier to output frames of 10,000 samples that mfcc must convert to frames of 128 samples. To handle this mismatch, the compiler must introduce a Converter that receives frames of 10,000 samples and outputs frames of 128 samples. Since 10,000 is not a multiple of 128, the Converter cannot be implemented efficiently as it requires at least some samples to be copied. In contrast, if the rmsClassifier were to output a frame of 10,240 samples, then the samples can be divided into 80 vectors that contain 128 samples as required by mfcc. This may be implemented without copying by defining 80 non-overlapping views over the same memory buffer containing the 10,240 samples.

The goal of the flow analysis is to find a solution to the type materialization problem that may be implemented efficiently. The flow analysis is performed at compile time and, therefore, it does not introduce any run-time overhead. The analysis is performed on the application SFG (see Figure 3 for an example) by considering each path sequentially. A path in the SFG captures the flow of frames from a source to a tap following internal and external links. For example, the frames from audio follow two paths: audio $\rightsquigarrow$ $\mathtt{T}_1$ and audio $\rightsquigarrow$ $\mathtt{T}_2$. The behavior of a conversion is determined by three variables: super-frames (S), frames (F), and multipliers (M). A super-frame is a contiguous block of memory that can be divided into an integer number of frames. Components along all paths that originate at a source s use super-frames of the same size (S_s). Each port p of a component A may require different frame sizes $F_{A,p}$. This requirement is fulfilled by having A execute M_A times to ensure $S_s = F_{A,p} \cdot M_A$. When these constraints are satisfied, all conversions on the paths from s may be implemented efficiently.

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 4. For clarity, we consider the case of determining the appropriate conversions for a single source that has a super-frame of size S. The algorithm considers each port p of a component A on the path. Let $C_{A,p}$ be the set of type constraints associated with port p of component A. A constraint has the form $(\bowtie, v)$, where $\bowtie$ is an operator ($\bowtie \in \{<, \leq, =, \geq, >\}$) and v is an integer. The algorithm iterates through each type constraint, adding new constraints to the ILP problem. If $\bowtie \in \{=\}$, then the size of the frame ($F_{A,p}$) is set to equal v (as specified by the type constraint) and we ensure that the super-frame (S) is a multiple

```
1:for $(A,p)$ a component-port pair path
2:   for $(\bowtie, v) \in C_{A,p}$:
3:       ILP: $0 < F_{A,p} \leq S$
4:       ILP: $M_A \geq 1$
5:       hasEquals = False
6:       if $\bowtie \in \{=\}$:
7:            hasEquals = True
8:            ILP:  $F_{A,p} = v$
9:            ILP:  $S = F_{A,p} \times M_A$
10:      elif $\bowtie \in \{\leq, <, >, \geq\}$:
11:           ILP:  $F_{A,p} \bowtie v$
12:  if hasEquals = False:
13:       ILP:  $M_A = 1$
14:       ILP:  $S = F_{A,p} \times M_A$
```

Fig. 4. Frame analysis: Algorithm and notation

of $F_{A,p}$ (lines 6 – 9). The multiplier M_A is optimized based the constraints of entire path. If the user does not supply a "=" constraint ($\bowtie \notin \{=\}$) (lines 13 – 14), then we set $M_A = 1$ indicating that the component can process the entire super-frame in a single call. In this case, the value of the frame $F_{A,p}$ will be optimized based on the constraints of the entire path. If $\bowtie \in \{\leq, <, >, \geq\}$ (lines 10 – 11), then the size of the frame ($F_{A,p}$) is constrained by v. Additionally, the frames sizes $F_{A,p}$ are constrained to be smaller or equal to then super-frame sizes S (line 3) and multipliers (M_A) to be at least 1 (line 4).

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 typical MSA has an associated ILP with multiple feasible solutions. Choosing an appropriate solution involves a trade-off between memory utilization and run-time overhead. CSense currently uses the solution that has the least memory utilization. This is because Android imposes strict limits on the memory utilization of an application, which limits an effective evaluation of trade-offs in selecting different solutions.

The ILP does not have a feasible solution 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 indicating that inefficient conversions are used and then 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.

F. Compiler

The compiler has the following workflow. The main configuration instantiates components, and configures, and connect them. 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 `Converters`, as appropriate. The SFG is partitioned into domains and then checked for race conditions. The final step is to generate code for the target platform.

We use 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. 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 applications on both Linux and OS X.

III. Run-time Environment

A. 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 follows. Each source maintains a memory pool that contains a number of super-frames. A source retrieves a super-frame from the pool when it has data to write. Flow analysis (performed at compile time) 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. During the initialization of an application, a configurable number of frames (currently set to 8) are preallocated in each memory pool. Additional frames may be allocated at run-time when new frames are request but none are available in the pool. These mechanisms limit 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 no later than 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 of schedulers and memory pools of sources. However, the underlying implementation of these data structures uses 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 contention case. If the lock implemented using atomic variables is not acquired after several attempts, we switch to using reentrant locks.

B. 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 consumption.

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 $t_{first} < t_{now}$, 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 $t_{first} \geq t_{now}$, 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 wake up 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 an 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 of pending events due to sleep.

IV. Evaluation

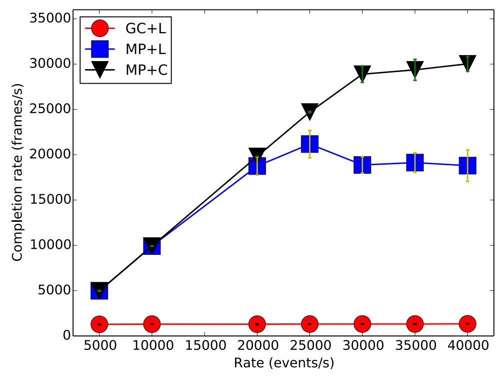
In this section, we provide an empirical evaluation of CSense on Galaxy Nexus phones running Android Jelly Bean. Galaxy Nexus uses a Texas Instruments OMAP 4460 SoC that includes a 1.2 GHz dual-core ARM Contex-A9. The phone has 1GB of memory and 32 GB of storage. C code is generated from MATLAB functions using MATLAB R2012b and MATLAB Coder 2.3. The resulting code is cross-compiled into a static library using Android NDK (r8d). We evaluate CSense using both micro- and macro-benchmarks.

A. Micro-benchmarks

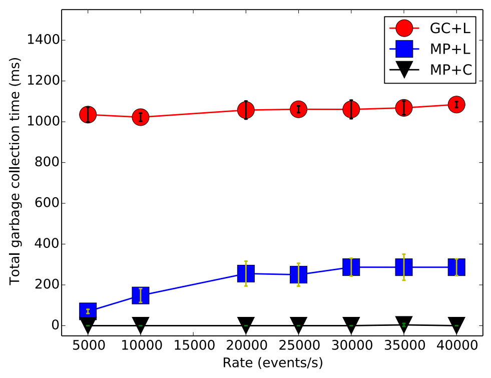
Scheduler Scalability: We evaluated the scalability of the scheduler using a Producer-Consumer benchmark. The producer generates frames at specified rates. The produced frames are passed to the consumer and then to a tap. The producer and consumer operate in different domains to capture the impact of 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 garbage collection is used to free them. Flow analysis is not used in this benchmark. 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. The results are averages over five runs; each run lasting for a minute. 95%-confidence intervals are also plotted.

Figure 5(a) shows the performance of the three schedulers. We increase the offered rate linearly and measure the rate at which the consumer receives frames. A scheduler should match the offered rate until it reaches its peak rate. To understand the differences in performance, 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.

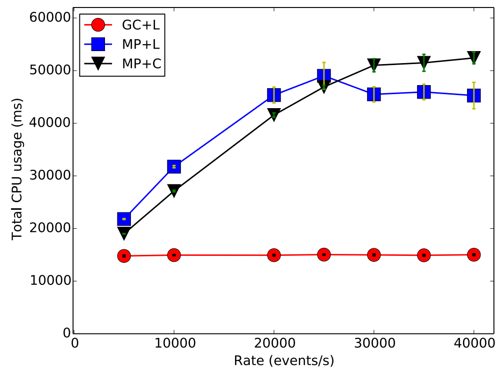
A naive implementation of the scheduler would use Java's memory management and locking concurrency primitives (`GC+L`). `GC+L` performs poorly; it supports a peak rate of only 1,534 events/s. `MP+L` incorporates memory pools and relies on locking concurrency primitives. Memory pools eliminate the creation of frames and reduce garbage collection. As a result, the peak event rate is increased to 21,176 events/s – a 13.8 times increase. Figure 5(b) plots the garbage collection time for each implementation as reported by Dalvik. As expected, the naive implementation has the highest garbage collection time. `MP+L` reduces garbage collection significantly, but does not eliminate it. In fact, the garbage collection time increases slowly with the offered rate. This increase may be attributed



(a) Scheduler implementations

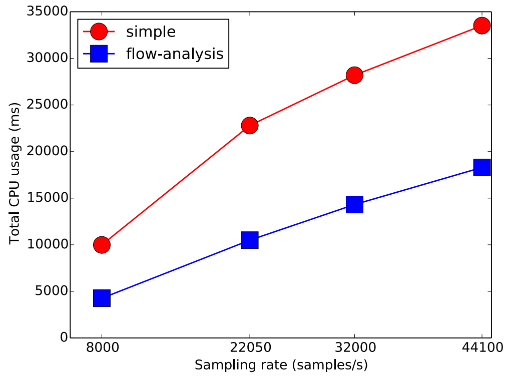


(b) Total garbage collection time

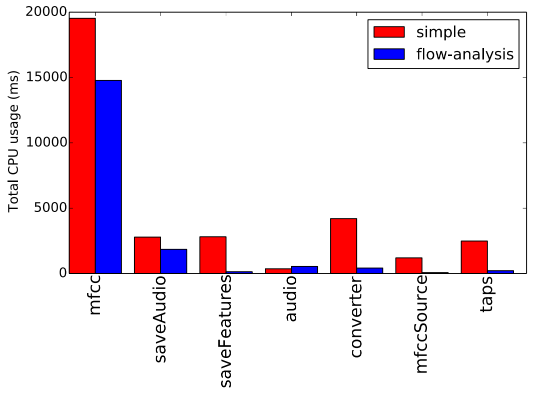


(c) CPU usage

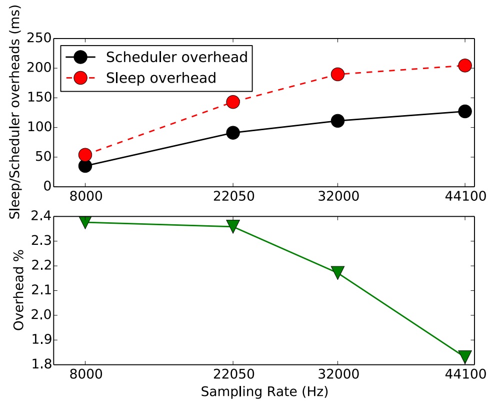
Fig. 5. Producer-consumer benchmark: Assessing the impact memory pools and concurrency mechanisms



(a) Benefits of flow analysis



(b) Detailed performance at 44100 Hz



(c) Scheduler overhead

Fig. 6. MFCC benchmark: Assessing the benefits of the flow analysis

to `ReentrantLock` objects being created in Java concurrent collections. These objects are created when a thread attempts to access a lock that is already held by a different thread.

Using our concurrency primitives, the scheduler (`MP+C`) may support a peak rate of 30,029 events/s. This represents an additional 30% improvement over `MP+L`. Overall, the proposed optimizations provide a 19 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 5(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. This allows for multiple frames to be inserted by the consumer or removed by the producer from the synchronization queue without requiring context switches.

B. Macro-benchmarks

We have implemented three applications to showcase the versatility of CSense: SpeakerIdentifier, ActiSense, and AudioSense. CSense facilitated incorporating MATLAB code in

Application	Number of constraints
SpeakerIdentifier	53
AudioSense	164
ActiSense (phone only)	172
ActiSense (phone + 3 Shimmer motes)	564

Fig. 7. Number of ILP constraints for each application

applications and its integration with additional Java components for data collection, file I/O, networking operations, and UI. Each application highlights an aspect of the toolkit: the benefits of flow analysis, the scheduler scalability, and its overhead. Statistics regarding the size of the ILP problem for each application and compilation times are also provided.

SpeakerIdentifier: SpeakerIdentifier (see Figure 3) determines the identity of speakers based on their voice fingerprint. The application is based on SpeakerSense [15]; however, in contrast to SpeakerSense, SpeakerIdentifier performs speaker identification remotely. The `mfcc` component is implemented in MATLAB. To evaluate the effectiveness of flow analysis on a realistic application, we will compile and profile the application with (`flow-analysis`) and without (`simple`) flow analysis.

Figure 6(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 6(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) Super-frames allow components to execute at different rates. The super-frame is 4096 samples, but the `mfcc` source and `saveFeature` 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 shorts to doubles. For efficiency, this component is implemented in native code. A benefit of using CSense is that the compiler can automate such transformations thus reducing the development burden.

We instrumented the scheduler to measure the time each domain thread spends executing the user code and the total time the thread was executed. The difference between the total and user time is considered overhead. We divide the overhead into sleep overhead and scheduler overhead. The sleep overhead measure the time to access the underlying power locks associated with each domain. Figure 6(c) plots detailed scheduler overhead when the flow analysis is used. The overhead percentage is computed relative to the total CPU usage reported in Figure 6(a)). The overhead ranges from 2.37% to 1.83%, decreasing slightly as the sampling rate increases. The slight decrease is the result of the CPU usage increasing faster than the scheduler overhead. These results show that the scheduler introduces small overhead.

Figure 7 includes the number of ILP constraints that were generated by the flow analysis for each application. All applications include less than a thousand linear constraints. ILP solvers can solve problems of this size very efficiently. On a laptop with a 2.6 GHz Intel Core i7 with 16 GB of RAM all ILPs were solved in less than 10 ms.

ActiSense: ActiSense is an activity recognition application that uses accelerometers to recognize running, sitting, walking, standing, and climbing stairs. The system is based on [2]. ActiSense includes a mobile phone and three Shimmer motes. Data from the Shimmer motes is streamed to the phone over Bluetooth. Feature extraction is performed on the phone. The extracted features are mean, time-domain and frequency-domain entropy, and correlation features as described in [2]. A Support Vector Machine (SVM) classifier determines user activities in real-time.

We have conducted a small user study involving 3 volunteers to evaluate the accuracy of the system. While instrumented, the volunteers performed the target activities as part of a circuit of activities. The collected data was annotated with the start and end times of each activity. The annotated data set was divided into training and testing data sets. We have evaluated four classifiers (Naive Bayes, ensembles of Naive Bayes classifiers, SVM, and ensembles of SVM classifiers) on the collected data set. The classification accuracy computed using 10-fold cross-validation is shown in Figure 8.

The components of ActiSense are partitioned into six domains. Four domains are responsible for collecting acceleration readings, saving them to flash, and making predictions using an SVM classifier. Three of the four domains are allocated for processing acceleration data from motes (one per mote). An additional domain is allocated to process acceleration data from the phone. The remaining two domains are responsible for recording data from the gyroscope and magnetometer sensors to disk.

Figure 9 plots the user time and associated overhead for each domain. The domains processing data from the Shimmer motes spent a similar amount of thread time – about 400 ms. The bulk of the time is spent on feature extraction (275 ms) and classification (68 ms). In contrast, the time to process the data collected from the phone's accelerometer is twice as long. The increase is reflected in a longer feature extraction (530 ms) and prediction (95 ms) times. An explanation for this difference is that the sensors use different sampling rates: the Shimmer motes are sampled at 50 Hz while the phones are sampled at 60 Hz. The overhead across all domains is about 13%. Half the overhead can be attributed to operations on Android's power locks. The higher overhead of ActiSense (compared to that of SpeakerIdentifier) is due to smaller super-frames. ActiSense was configured to produce activity predictions every second, which prevented the creation of large superframes. Relaxing this constraint will reduce overhead, as the flow analysis will use larger super-frames.

AudioSense: AudioSense [4] evaluates the performance of hearing aids using electronic surveys. Surveys may be user initiated or triggered at random intervals on average every 1.5 hours. Concurrent with the delivery of surveys, AudioSense collects audio samples and GPS locations to provide a context for the surveys. AudioSense caches data on the mobile phone

Classifier	Configuration	Accuracy
Naive Bayes	Phone + Shimmer	69.54%
Naive Bayes Ensemble	Phone + Shimmer	76.55 %
SVM	Shimmer	88.43%
SVM	Phone	94.64%
SVM Ensemble	Shimmer + Phone	96%

Fig. 8. ActiSense: Accuracy for different configuration and learning algorithms

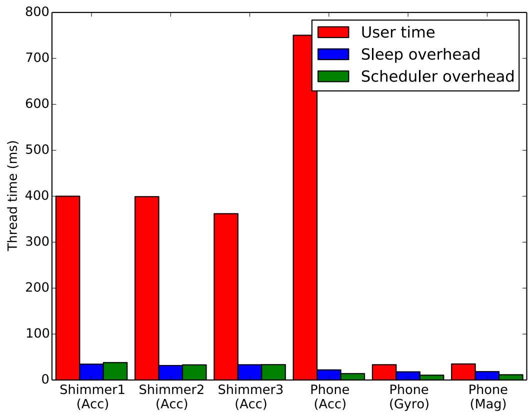
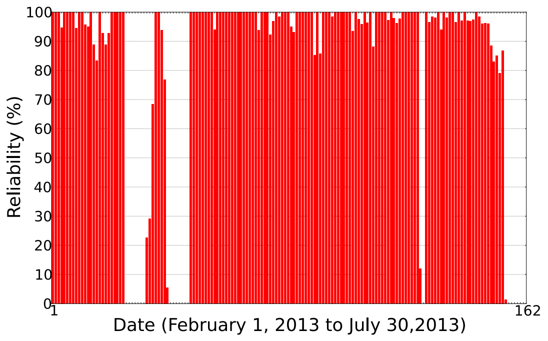


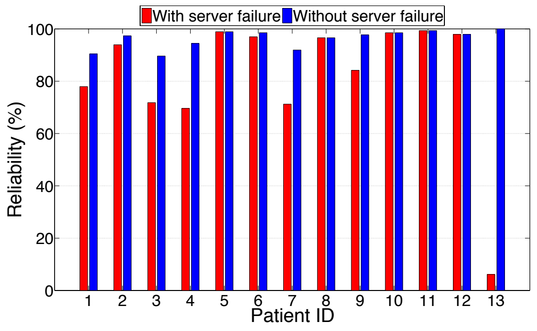
Fig. 9. ActiSense: Execution time and overheads

and uploads it when it may establish a 3G connection to our remote server. AudioSense has been deployed for six months as part of a clinical study. The challenge of developing such a system is to ensure reliability during weeklong deployments.

Figure 10 plots the reliability for each day of the trial. The reliability is computed as the fraction of data uploaded out of the data that was collected. As shown in the graph, there were three instances when the reliability was zero during the trial. The cause of these outages was the server being offline for several days due to power outages. The remainder of the outages may be attributed to coverage gaps in the study area. Figure 10(b) shows the reliability for the 13 patients in the study included in the first six months of the study. Excluding the server outages, the reliability of AudioSense exceeds 90%. This shows that our toolkit is sufficiently mature to support long-term deployments.



(a) Reliability per day



(b) Reliability per patient

Fig. 10. AudioSense: Reliability during a 6-month trial

V. Related Work

CSense uses a stream processing (SP) model to support the development of high-rate and robust MSAs. This section places CSense in the context of prior work on SP systems and static analysis of Java programs.

Stream Programming: SP models have been studied for decades (see [16] 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 [7], [16], [17]. Asynchronous systems provide a more flexible concurrency model but sacrifice performance, as many of the optimizations developed for synchronous systems do not translate these systems. CSense adopts an asynchronous model to support workloads that include both concurrent operations and asynchronous events.

The problem of efficiently supporting asynchronous SP has been previously considered in systems such as Click [5], XStream [18], and WaveScript [17]. Click executes components in a single thread but avoids creating inflexible fixed schedules. Click maintains a task queue to which sources and queue components are added when they have data to process. A scheduler determines the execution order of the tasks in the queue. The execution of the other type of components is triggered by function calls that traverse the component graph in a depth-first manner. A similar approach to component scheduling is used in XStream and WaveScript. As described in Section II-C, CSense extends this mechanism by including support for multiple execution domains and event handling. CSense further reduces overhead through a flow analysis that allows components to be executed multiple times without involving the scheduler.

Memory management can have a significant performance impact on SP. XStream and WaveScript use an abstract data structure called SigSegs to efficiently exchange frames between components. SigSegs have some similarity to our flow analysis that optimizes the memory allocation of frames. However, in contrast to SigSegs that operate solely at runtime, we optimize memory management by leveraging type information and explicit knowledge of frame flows at compile time. This optimization is feasible in CSense due to the additional information supplied by developers.

Static analysis: There have been several efforts to detect concurrency problems in Java programs. For example, ESC/Java2 [19] and Checker Framework [20] are static analysis tools that identify potential bugs such as data races. Unfortunately, these tools are neither sound nor complete, they cannot fully address the problems of aliasing and limited visibility into the Java/Android run-time environment. More promising results are obtained in domain specific languages. For example, NesC [21] limits programmers to using only static memory and a restricted concurrency model to facilitate static analysis. CSense adopts a similar strategy by explicitly capturing memory operations as part of component graphs and limiting its concurrency model.

VI. Conclusions

In this paper, we presented CSense – a stream programming toolkit for developing high rate and robust mobile sensing applications on Android. CSense provides developers a programming model, a compiler, and a run-time environment. The programming model extends existing SP models by incorporating a flexible concurrency model, a new type systems that fosters component reuse, and incorporates memory operations as part of the SFG. We leverage this additional information for both the compilation and static analysis. Our compiler incorporates a novel flow analysis that optimizes frames exchange across components from an application-wide perspective. Empirical results indicate that the flow analysis may reduce CPU utilization as much as 45%. Moreover, static analysis techniques can prevent a range of programming errors including the incorrect usage of the memory management system and data races. These techniques enabled us to deliver a mobile sensing application that uploads data to a server with over 90% reliability. We have identified that the memory management and concurrency limit the scalability of SP on Android. We incorporate memory pools, frame conversion optimizations, custom synchronization primitives, and careful integration with power locks to develop a scalable run-time environment. Micro-benchmarks indicate that these optimizations increase the peak stream rate by as much as 19 times over a baseline implementation that uses Java's concurrency management.

VII. Acknowledgments

This work supported by the National Science Foundation (Grant No. 1144664) and by the Roy J. Carver Charitable Trust (Grant No. 14-4355).

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," R. N. Horspool, Ed., vol. LNCS 2304, 2002.

[8] 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.

[9] 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.

[10] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song, "SeeMon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments," *MobiSys*, 2008.

[11] S. Kang, Y. Lee, C. Min, and Y. Ju, "Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments," in *PerCom*, 2010.

[12] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of MobiSys*, 2010.

[13] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao, "Balancing energy, latency and accuracy for mobile sensor data classification," *SenSys*, p. 54, 2011.

[14] S. Nirjon, R. F. Dickerson, P. Asare, Q. Li, D. Hong, J. A. Stankovic, P. Hu, G. Shen, and X. Jiang, "Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones," in *MobiSys*, 2013.

[15] H. Lu, A. Bernheim Brush, B. Priyantha, A. Karlson, and J. Liu, "SpeakerSense: energy efficient unobtrusive speaker identification on mobile phones," *Pervasive Computing*, 2011.

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

[17] 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.

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

[19] ESC/Java2, "http://goo.gl/1wbvva."

[20] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *ISSTA*, 2008.

[21] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *PLDI*, 2003.