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

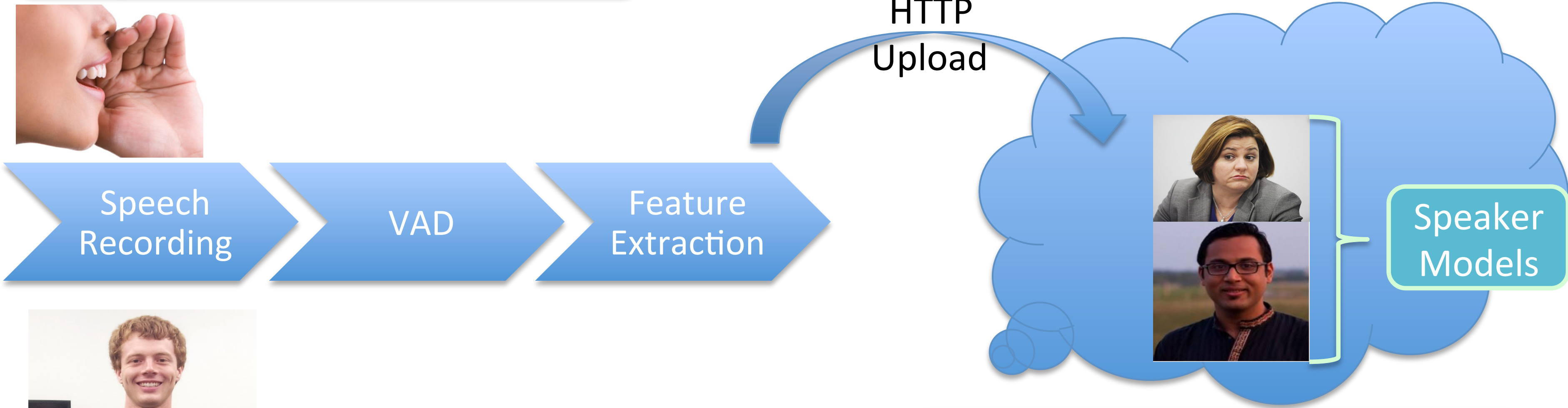
IPSN 2014

Farley Lai, Syed Shabih Hasan, Austin Laugesen, Octav Chipara
Department of Computer Science

THE UNIVERSITY OF IOWA

Mobile Sensing Applications (MSAs)

Speaker Identification



Activity Recognition

Sitting
Standing
Walking
Running
Climbing Stairs
…

Challenges

- Mobile sensing applications are difficult to implement on Android devices
  - concurrency
  - high frame rates
  - robustness

- Resource limitations and Java VM worsen these problems
  - additional cost of virtualization
  - significant overhead of garbage collection

Related Work

- Support for MSAs
  - SeeMon, Coordinator: constrained queries
  - JigSaw: customized pipelines

➔ CSense provides a high-level stream programming abstraction general and suitable for a broad range of MSAs

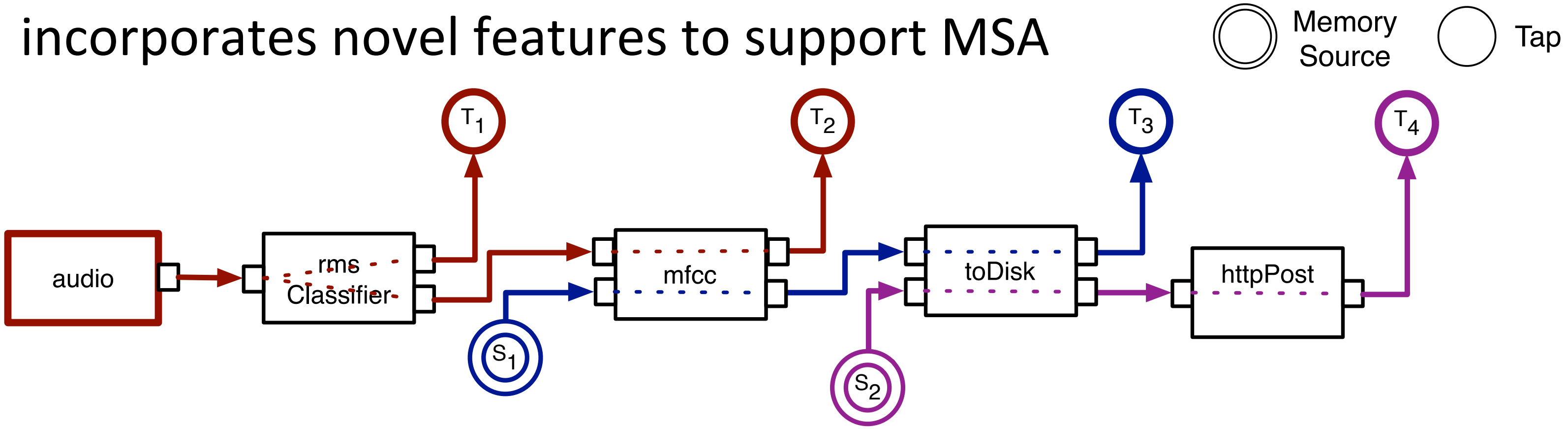
- CSense builds on prior data flow models
  - Synchronous data flows: static scheduling and optimizations
    - e.g., StreamIt, Lustre
  - Async. data flows: more flexible but have lower performance
    - e.g., Click, XStream/Wavescript

CSense Toolkit

- Programming model
- Compiler
- Run-time environment
- Evaluation

Programming Model

- Applications modeled as Stream Flow Graphs (SFG)
  - builds on prior work on asynchronous data flow graphs
  - incorporates novel features to support MSA



```
addComponent("audio", new AudioComponentC(rateInHz, 16));
addComponent("rmsClassifier", new RMSClassifierC(rms));
addComponent("mfcc", new MFCCFeaturesG(speechT, featureT))
...
link("audio", "rmsClassifier");
toTap("rmsClassifier::below");
link("rmsClassifier::above", "mfcc::sin");
fromMemory("mfcc::fin");
...
```

create components

wire components

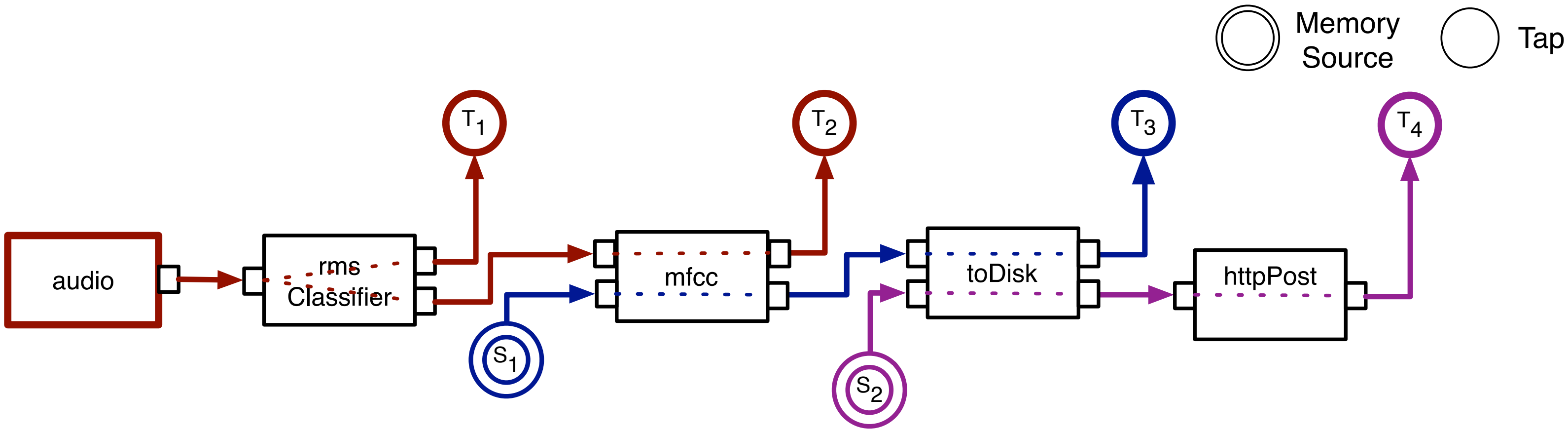
Memory Management

- Goal: Reduce memory overhead introduced by garbage collection and copy operations

- Pass-by-reference semantics
  - allows for sharing data between components

- Explicit inclusion of memory management in SFGs
  - focuses programmer's attention on memory operations
  - enables static analysis by tracking data exchanges globally
  - allows for efficient implementation

Memory Management



- Data flows from sources, through links, to taps

- Implementation:
  - sources implement memory pools that hold several frames
  - references counters used to track sharing of frames
  - taps decrement reference counters

Concurrency Model

- Goal: Expressive concurrency model that may be analyzed statically

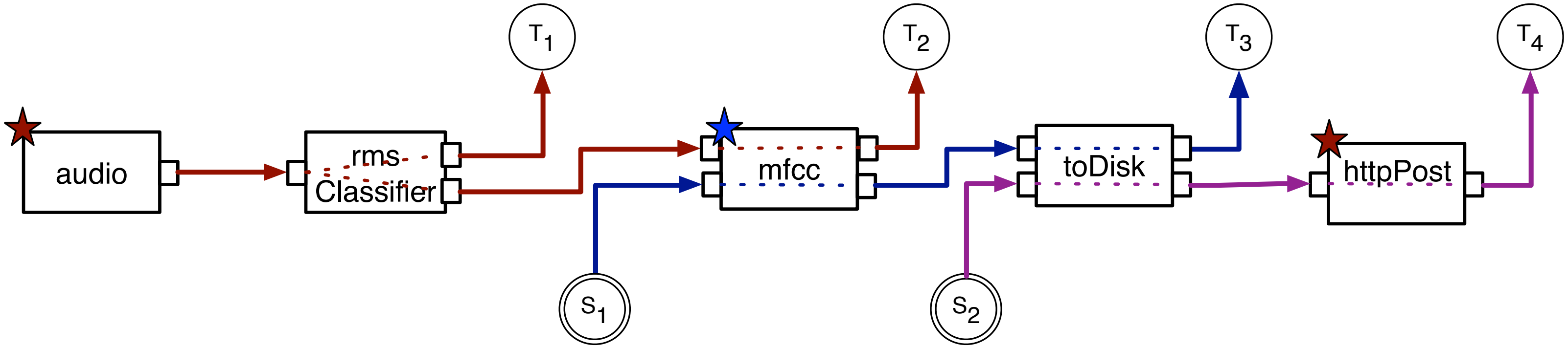
- Components are partitioned into execution domains
  - components in the same domain are executed on a thread
  - frame exchanges between domains are mediated using shared queues

- Other data sharing between components are using a tuple space

- Concurrency is specified as constraints
  - **`NEW_DOMAIN / SAME_DOMAIN`**
  - heuristic assignment of components to domains to minimize data exchanges between domains

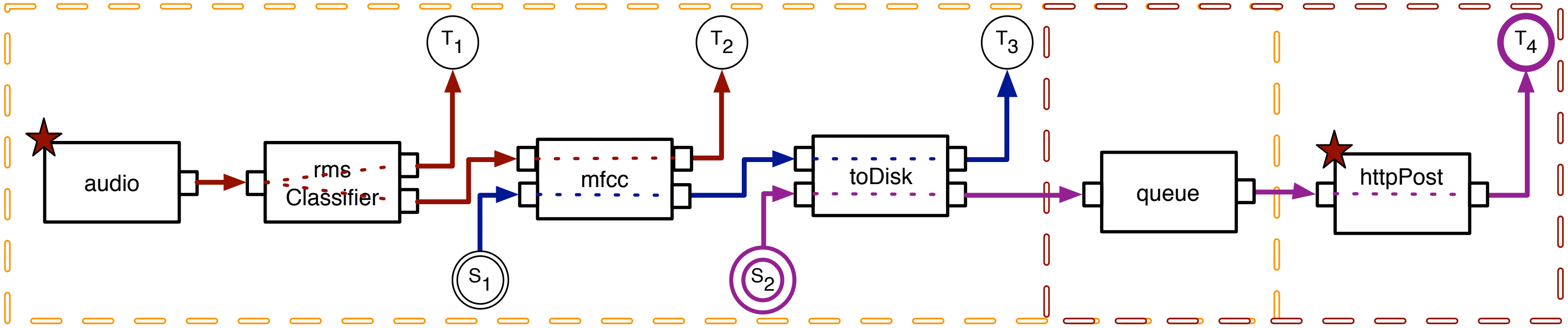
- Static analysis may identify some data races

Concurrency Model



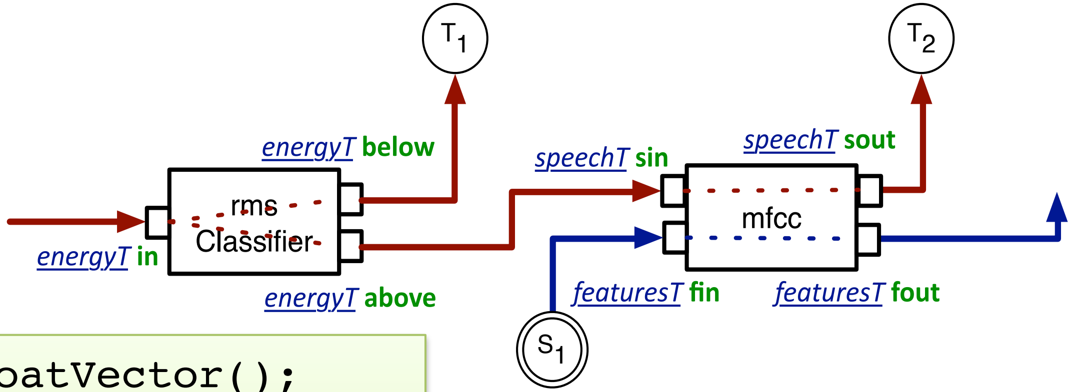
```
getComponent("audio").setThreading(Threading.NEW_DOMAIN);
getComponent("httpPost").setThreading(Threading.NEW_DOMAIN);
getComponent("mfcc").setThreading(Threading.SAME_DOMAIN);
```

Compiler transformation



Type System

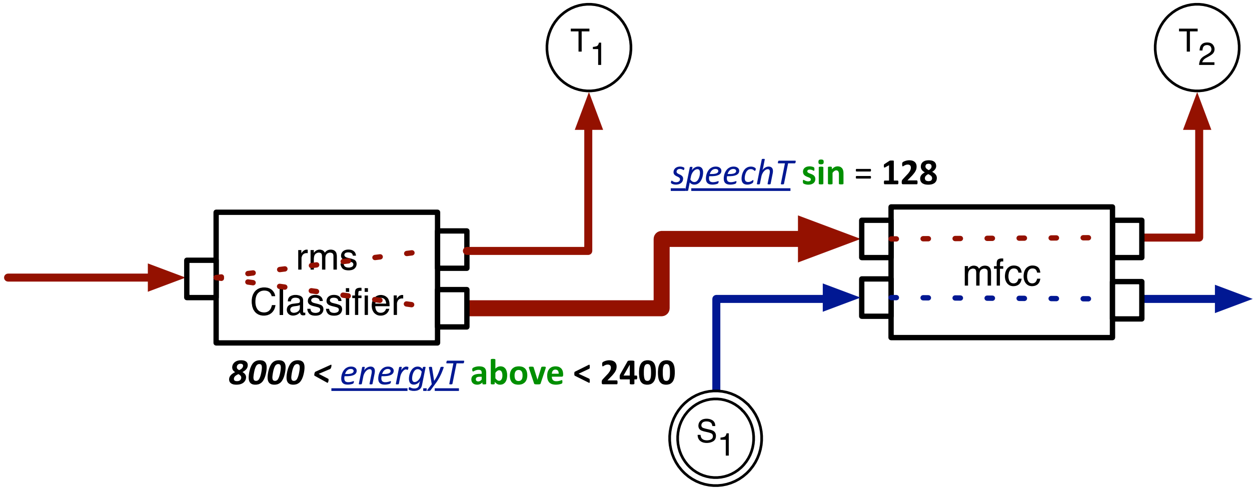
- Goal: Promote component reuse across MSAs
- A rich type system that extends Java's type system
  - most components use generic type systems
  - insight: frame sizes are essential in configuring components
    - detect configuration errors / optimization opportunities



```
VectorC energyT = TypeC.newFloatVector();
energyT.addConstraint(Constraint.GT(8000));
energyT.addConstraint(Constraint.LT(24000));
VectorC speechT  = TypeC.newFloatVector(128);
VectorC featureT = TypeC.newFloatVector(11);
```

Flow Analysis

- Not all configurations may be implemented efficiently

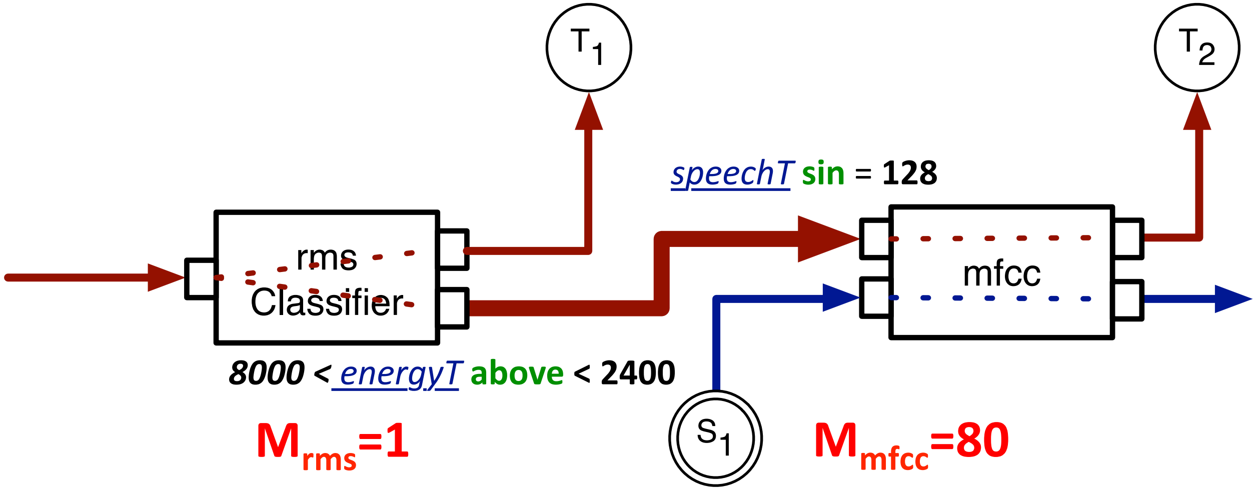


Constraints:
energyT > 8000
energyT < 24000
speechT = 128
featuresT = 11

	energyT	speechT
Inefficient	10,000	128
Efficient	10,240 (128 * 80)	128

Flow Analysis

- Not all configurations may be implemented efficiently



Constraints:
energyT > 8000
energyT < 24000
speechT = 128
featuresT = 11

	energyT	speechT
Inefficient	10,000	128
Efficient	10,240 (128 * 80)	128

An efficient implementation exists when
$M_{rms} * energyT = M_{mfcc} * speechT$

Flow Analysis

- Goal: determine configurations have efficient frame conversions

- Problem may be formulated as an integer linear program
  - *constraints:* generated from type constraints
  - *optimization:* minimize total memory usage
  - *solution:* specifies frame sizes and multipliers for application

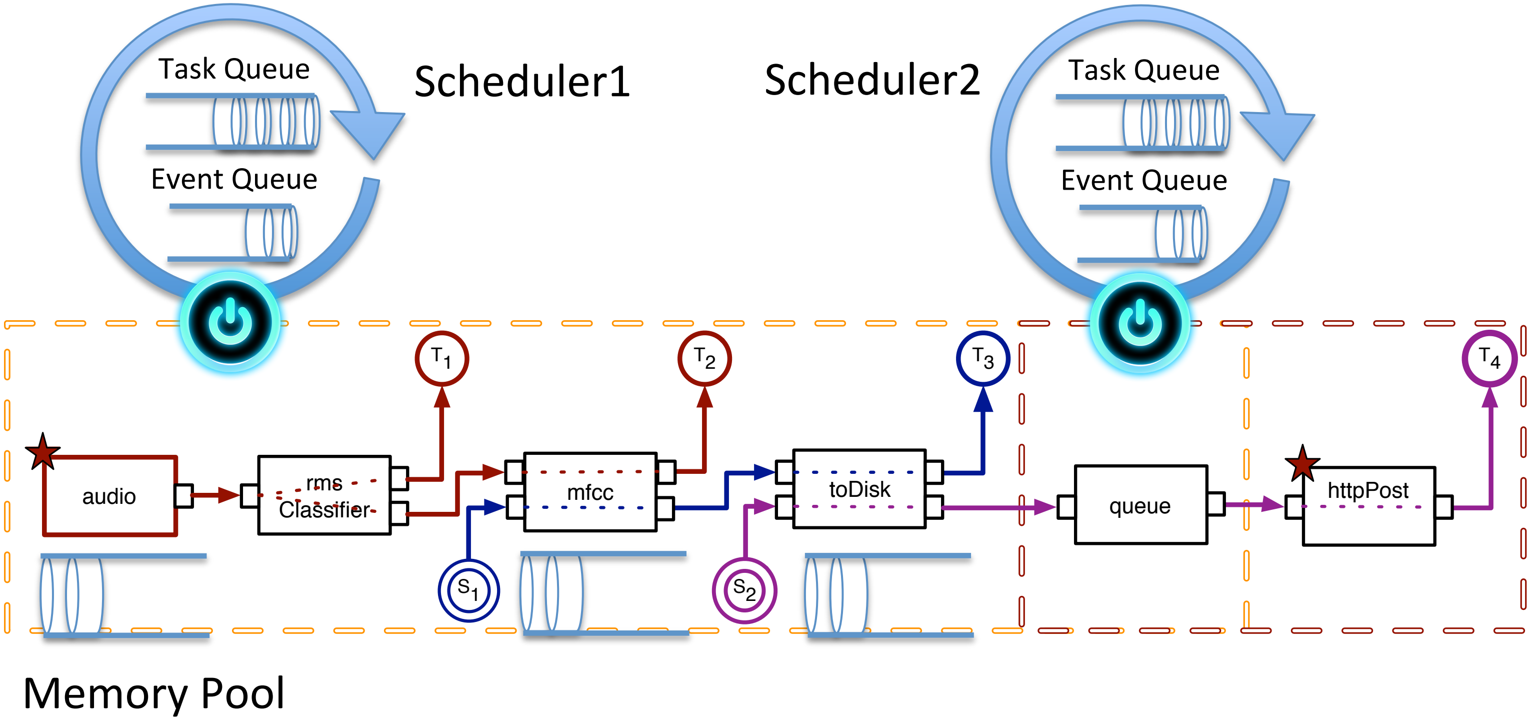
- An efficient frame conversion may not exist
  - the compiler relaxes conversion rules

CSense Compiler

- Static analysis:
  - composition errors, memory usage errors, race conditions
- Flow analysis:
  - whole-application configuration and optimization
- Stream Flow Graph transformations:
  - domain partitioning, type conversions, MATLAB component coalescing
- Code generation:
  - Android application/service, MATLAB (C code + JNI stubs)

CSense Runtime

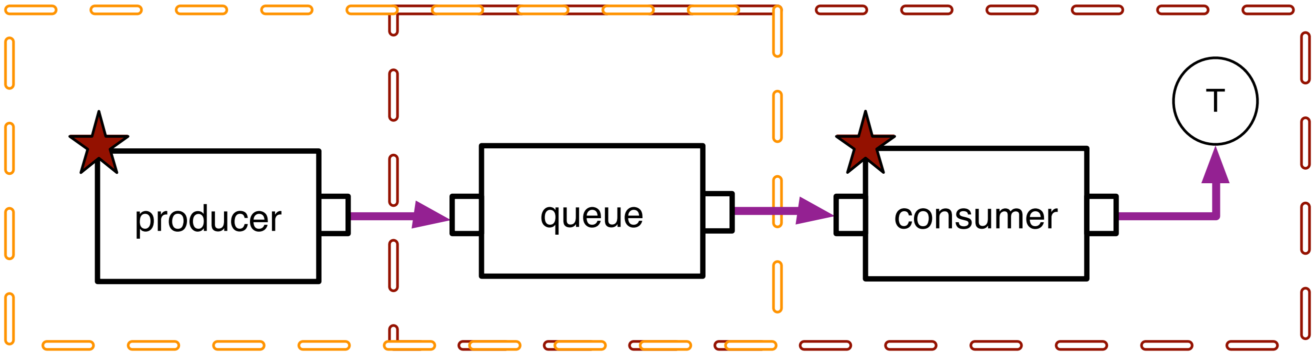
- Components exchange data using push/pull semantics
- Runtime includes a scheduler for each domain
  - task queue + event queue
  - wake lock – for power management



Evaluation

- Micro benchmarks evaluate the runtime performance
  - synchronization primitives + memory management
- Implemented the MSA using CSense
  - Speaker identification
  - Activity recognition
  - Audiology application
- Setup
  - Galaxy Nexus, TI OMAP 4460 ARM A9@1.2 GHz, 1 GB
  - Android 4.2
  - MATLAB 2012b and MATLAB Coder 2.3

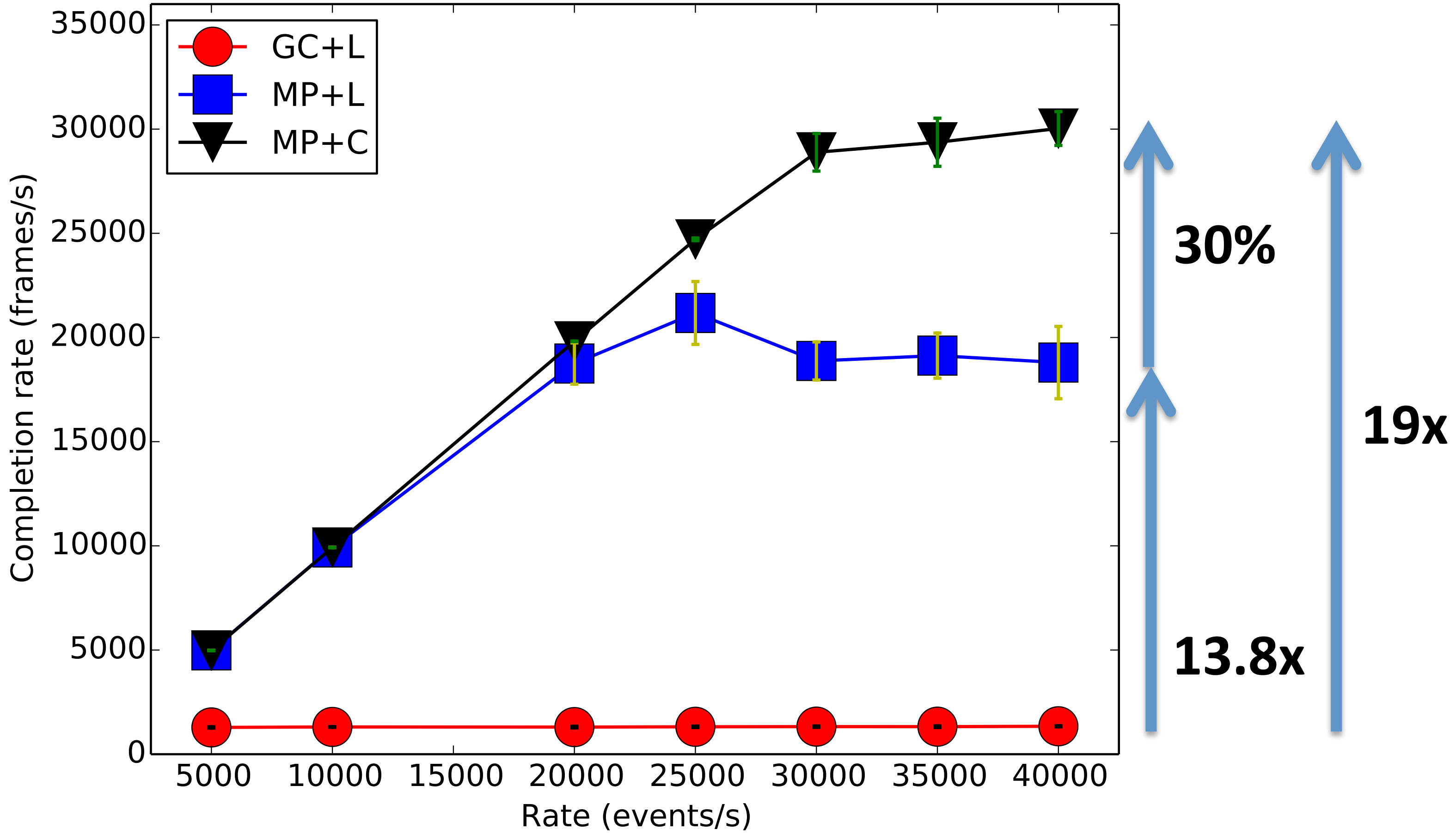
Producer-Consumer Benchmark



- Scheduler: memory management + synchronization primitives
- Memory management options
  - *GC*: garbage collection
  - *MP*: memory pool
- Concurrent access to queues and memory pools
  - *L*: Java reentrant lock
  - *C*: CSense atomic variable based synchronization primitives

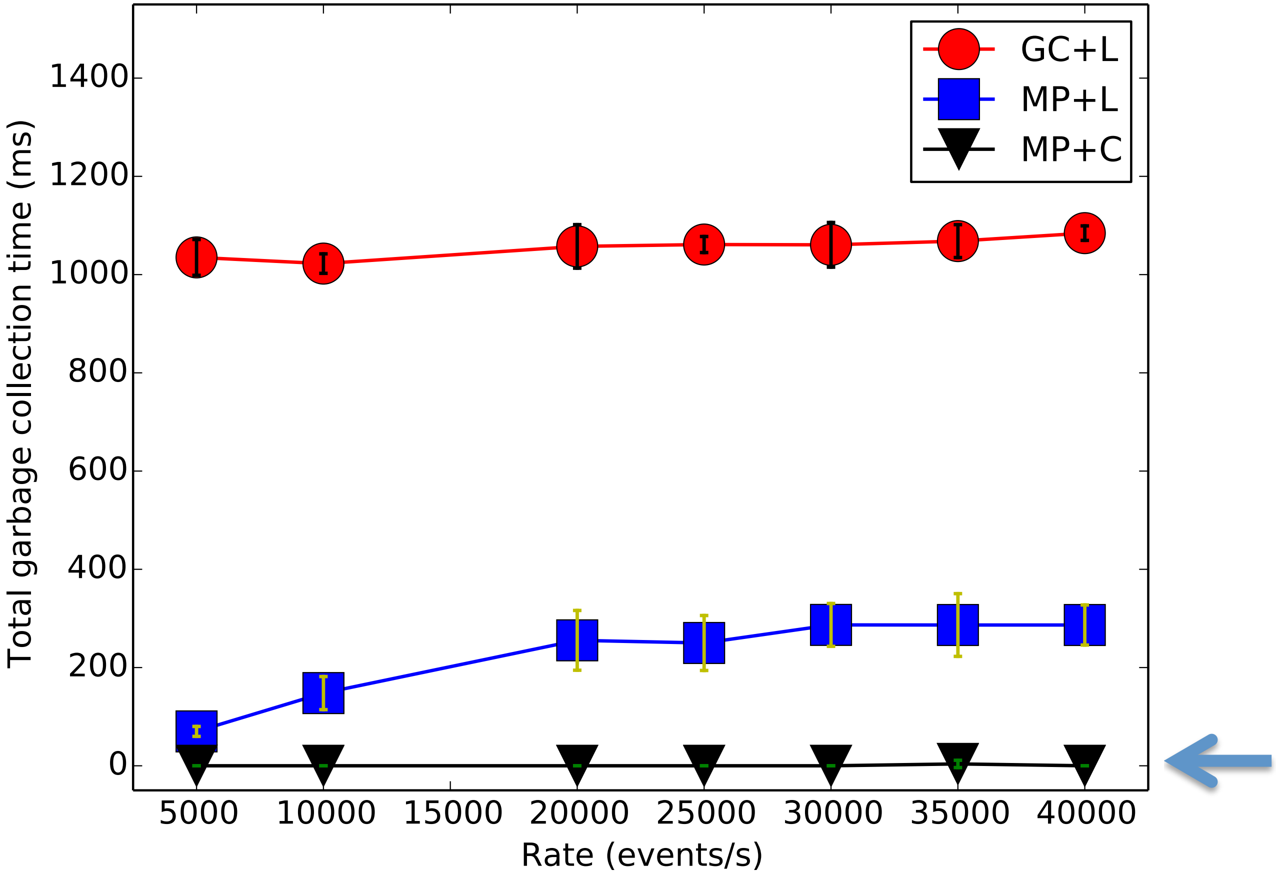
Producer-Consumer Throughput

- Garbage collection overhead limits scalability
- Concurrency primitives have a significant impact on performance



Producer-Consumer GC Overhead

- Reentrant locks incurs GC due to implicit allocations
- CSense runtime has low garbage collection overhead

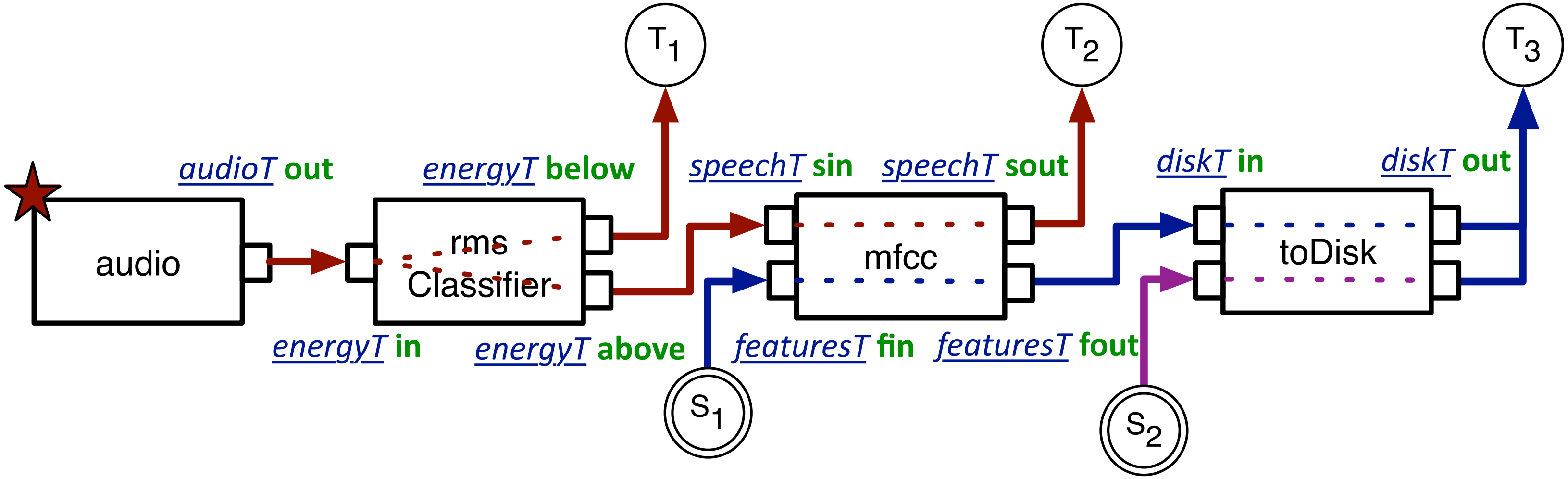


no garbage collection

(in this benchmark)

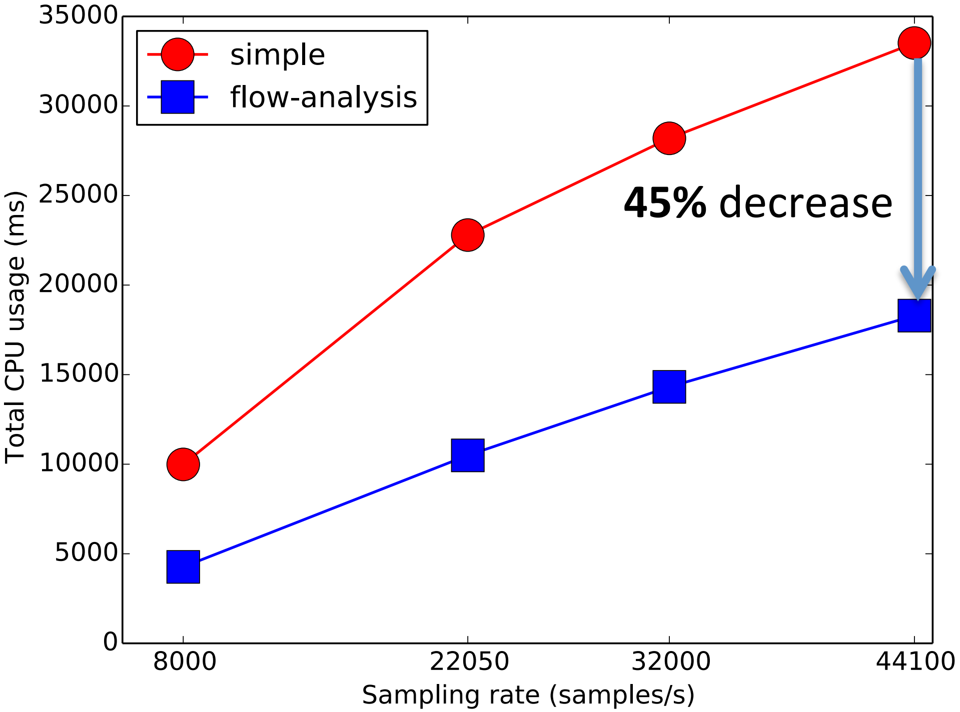
MFCC Benchmark

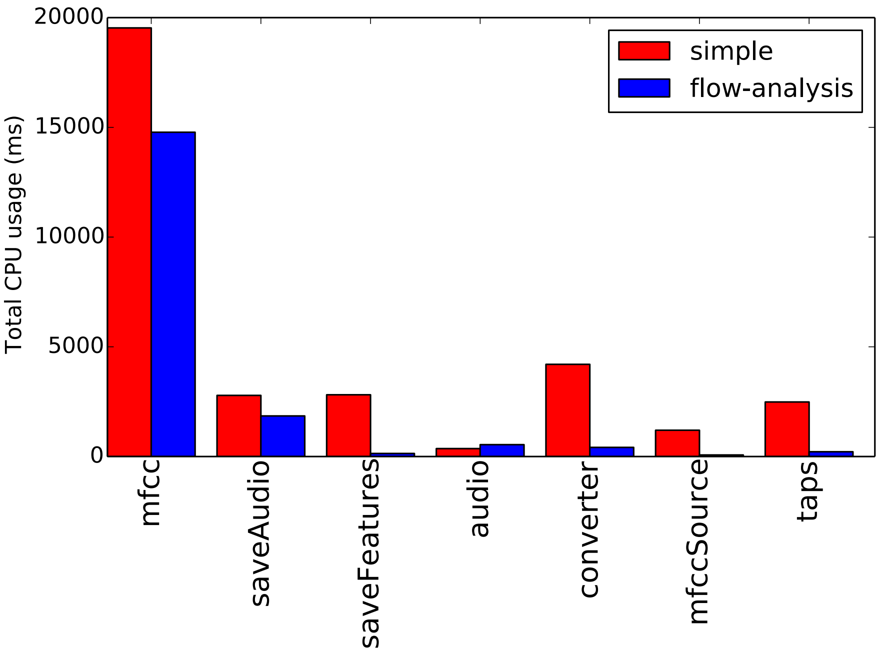
- Benefits of flow analysis
- Runtime overhead



MFCC Benchmark CPU Usage

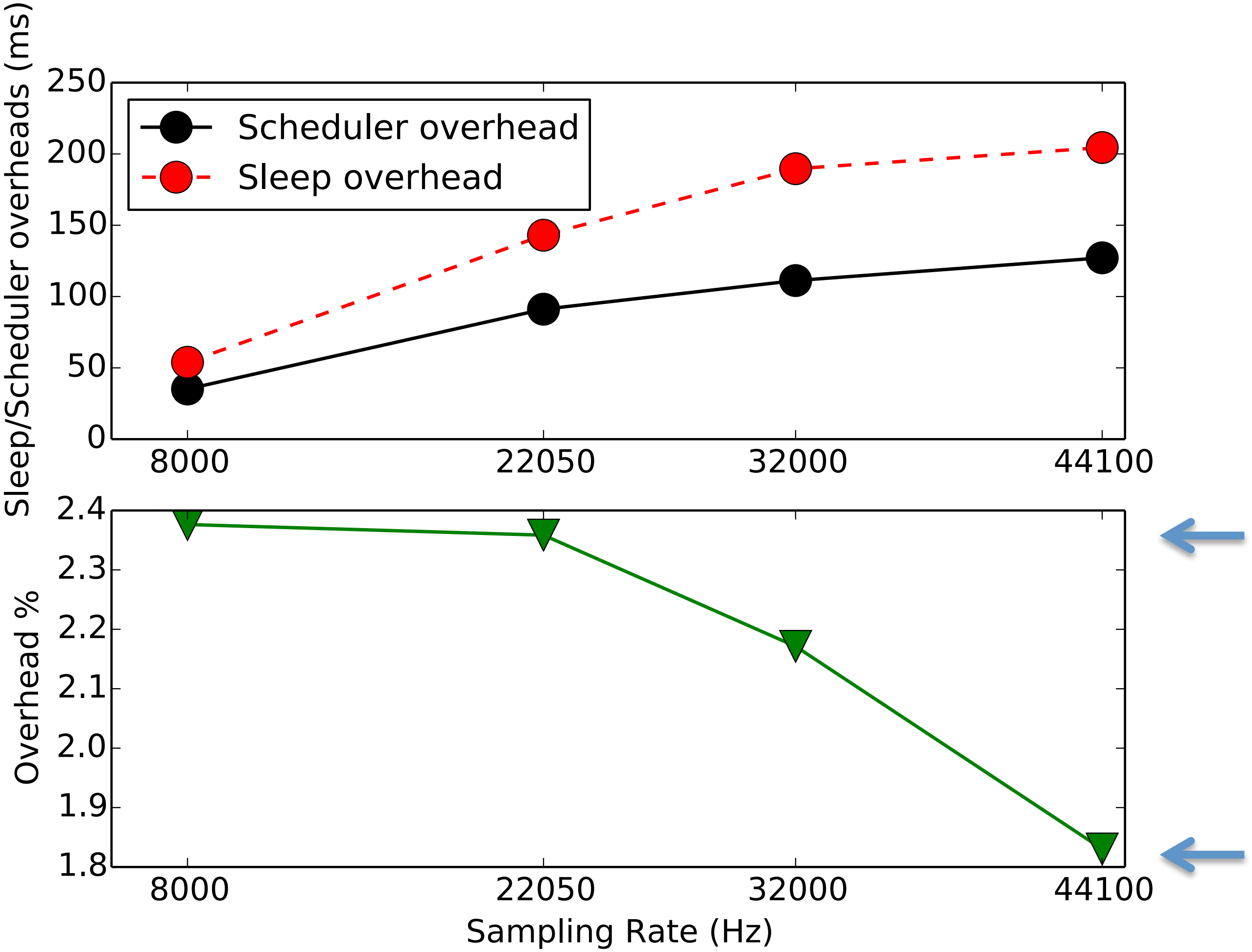
- Flow analysis eliminates unnecessary memory copy
- Benefits of larger but efficient frame allocations
  - reduced number of component invocations and disk I/O overhead


45% decrease



MFCC Runtime Overhead

- Runtime overhead is low for a wide range of data rates



2.39%

1.83%

Conclusions

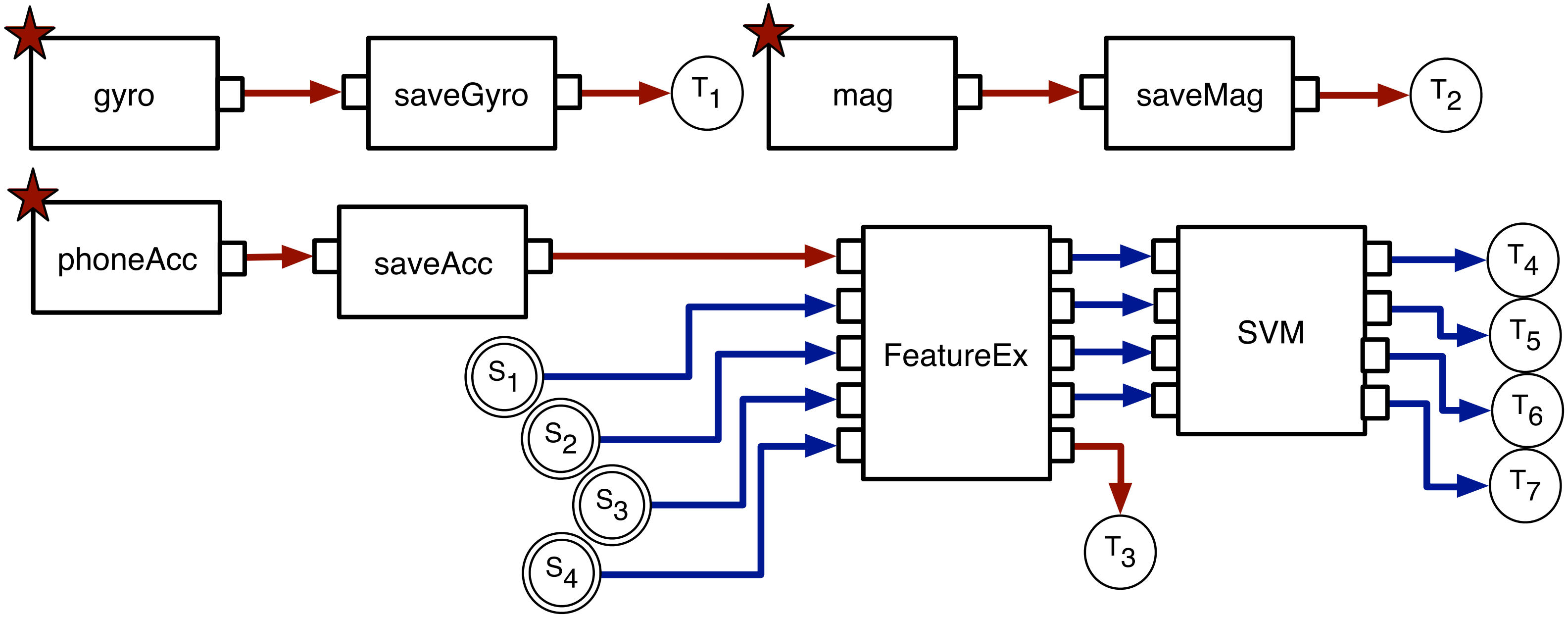
- Programming model
  - efficient memory management
  - flexible concurrency model
  - rich type system
- Compiler
  - whole-application configuration & optimization
  - static and flow analyses
- Efficient runtime environment
- Evaluation
  - implemented three typical MSAs
  - benchmarks indicate significant performance improvements
    - 19X throughput boost compared with naïve Java baseline
    - 45% CPU time reduced with flow analysis
    - Low garbage collection overhead

Acknowledgements

- National Science Foundation (NeTs grant #1144664)
- Carver Foundation (grant #14-43555)

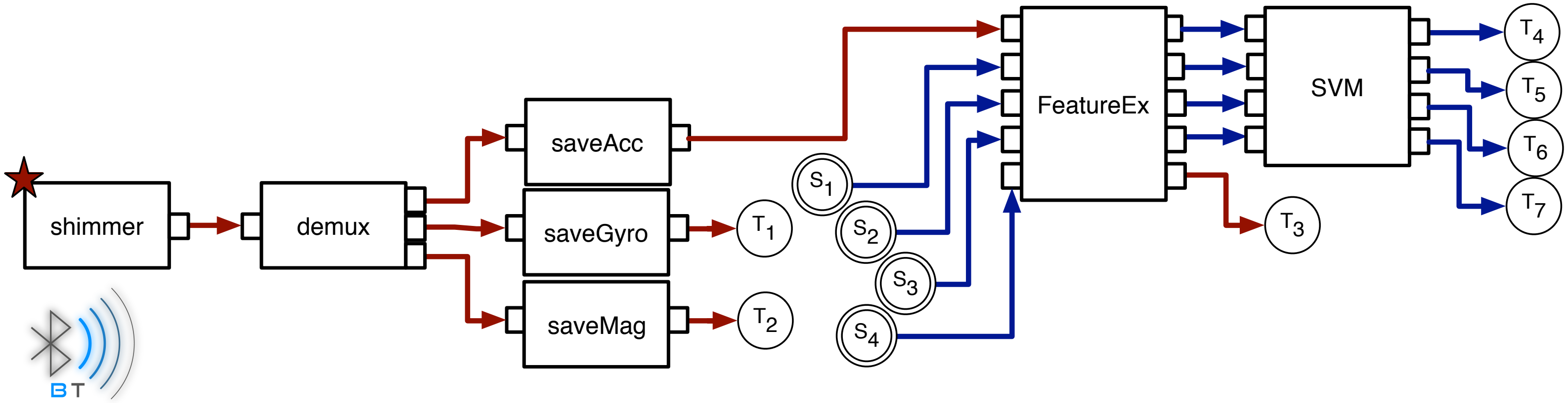
ActiSense Benchmark

- Runtime scheduler overhead of a complex 6-domain application that accesses both phone sensors and remote shimmer motes over bluetooth



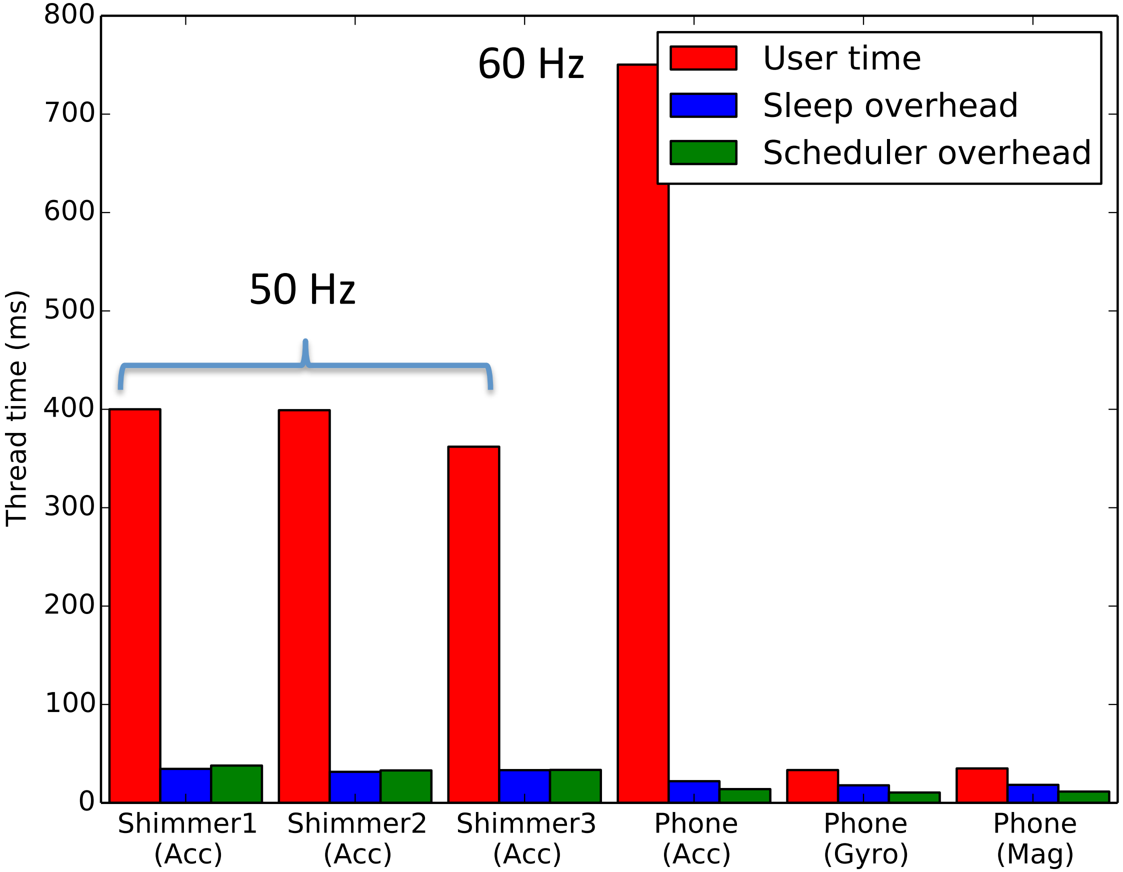
ActiSense Benchmark

- Runtime scheduler overhead of a complex 6-domain application that accesses both phone sensors and remote shimmer motes over bluetooth

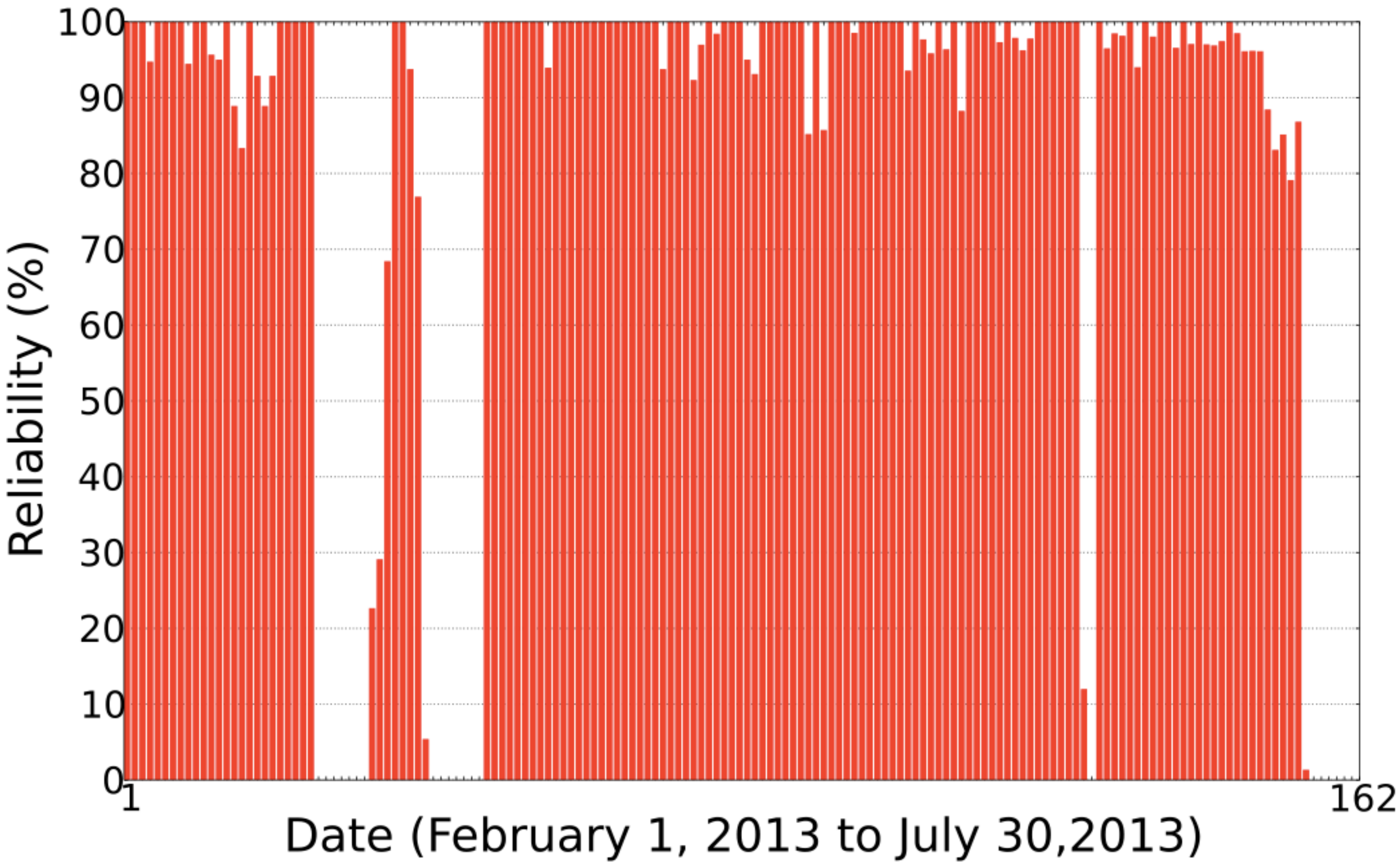


ActiSense CPU Usage

- Overall domain scheduler overhead is small despite a longer pipeline

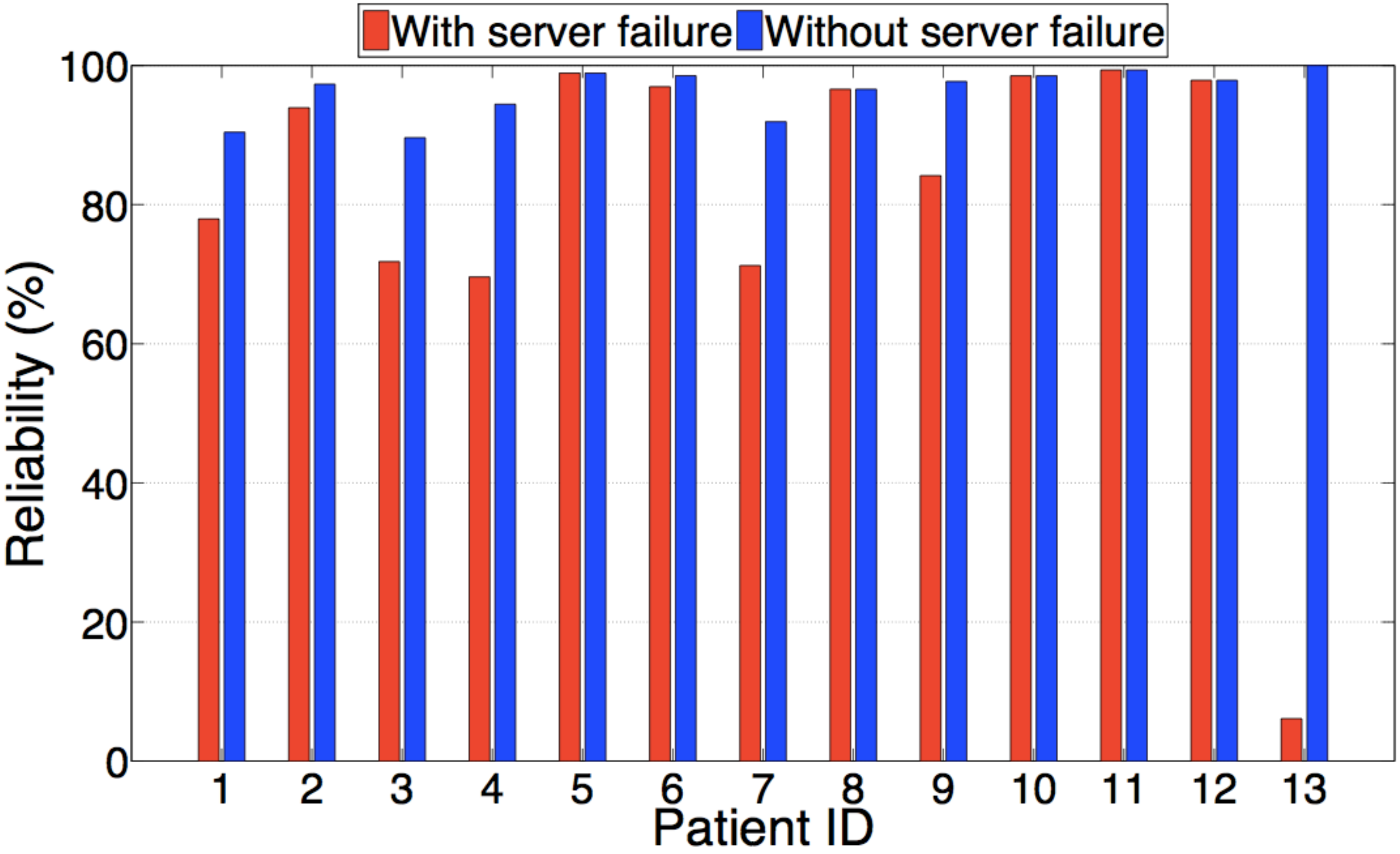


AudioSense



(a) Reliability per day

AudioSense



(b) Reliability per patient