A Case for Ending Monolithic Apps for Connected Devices

Rayman Preet Singh (*Univ. of Waterloo*) Chenguang Shen (*UCLA*)

Amar Phanishayee Aman Kansal Ratul Mahajan

Microsoft Research

1 Introduction

Connected sensing devices, such as cameras, thermostats, in-home motion, door-window, energy, water sensors [1], collectively dubbed as the *Internet of Things* (IoT), are rapidly permeating our living environments [2], with an estimated 50 billion such devices in use by 2020 [8]. They enable a wide variety of applications spanning security, efficiency, healthcare, and others. Typically, these applications collect data using sensing devices to draw inferences about the environment or the user, and use the inferences to perform certain actions. For example, Nest [10] uses motion sensor data to infer home occupancy and adjusts the thermostat.

Most applications that use connected devices today are built in a monolithic way. That is, applications are tightly coupled to the hardware, and need to implement all the data collection, inferencing, and user functionality related logic. For application developers, this complicates the development process, and hinders broad distribution of their applications because the cost of deploying their specific hardware limits user adoption. For end users, each sensing device they install is limited to a small set of applications, even though the hardware capabilities may be useful for a broader set of applications. How do we break free from this monolithic and restrictive setting? Can we enable applications to be programmed to work seamlessly in heterogeneous environments with different types of connected sensors and devices, while leveraging devices that may only be available opportunistically, such as smartphones and tablets?

To address this problem, we start from an insight that many inferences required by applications can be drawn using multiple types of connected devices. For instance, home occupancy can be inferred using motion sensors (such as those in security systems or Nest [10]), cameras (e.g. Dropcam [3], Simplicam [14]), microphone, smartphone GPS, or using a combination of these, since each may have different sources of errors. *We posit that inference logic, traditionally left up to applications, ought to be abstracted out as a system service.* Such a service should relieve application developers of the burden of implementing and training commonly used inferences. It should enable applications to work using any of the sensing devices that the shared inference logic supports.

This paper discusses the key challenges in designing the proposed service. First, the service must decouple i) "what is sensed" from "how it is sensed", and ii) "what is inferred" from "how it is inferred". In doing so, it should ensure inference extensibility, and provide a uniform interface to inferences which hides sensor-specific intricacies (e.g., camera frame rate, motion sensitivity level). Second, the service should enable seamless use of sensors on mobile devices by handling environmental dynamics resulting from device and user mobility. Third, while serving required inferences to applications, the service should ensure efficient use of resources, e.g., by selecting the optimal subset of sensors to serve active applications, sharing overlapping inference computations, and hosting computations on suitable devices.

To explore these challenges concretely, we propose **Beam**, an application framework and runtime for inference driven applications. Beam provides applications with inference-based abstractions. Applications simply specify their inference requirements, while Beam bears the onus of identifying the required sensors, initiating data processing on suitable devices, handling environmental dynamics, and using resources efficiently.

2 Key Design Requirements

Our motivation for designing Beam are data-driven-inference based applications, aimed at homes [10, 16], individual users [9, 11, 41, 48, 50] and enterprises [6, 13, 20, 33, 42]. We identify three key challenges for Beam by analyzing two popular application classes in detail, one that infers environmental attributes and another that senses an individual user.

Rules: A large class of popular applications is based on the 'If This Then That (IFTTT)' pattern [7, 47]. IFTTT enables users to create their own rules connecting sensed attributes to desired actions. We consider a particular rules application which alerts a user if a high risk appliance, e.g., electric oven, is left on when the home is unoccupied [15, 44]. This application uses the appliance-

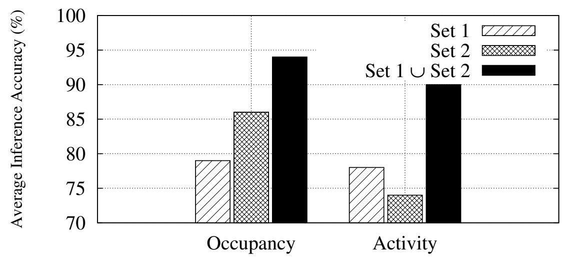

Figure 1: Improvement in occupancy and activity inference accuracy by combining multiple devices. For occupancy, sensor set 1 = {camera, microphone} in one room and set 2 ={PC interactivity detection} in a second room. For physical activity, set 1 = {phone accelerometer} and set 2 = {wrist worn FitBit [4]}.

state and home occupancy inferences.

Quantified Self (QS) [9, 11, 19, 25, 38, 46] which disaggregates a user's daily routine, by tracking her physical activity (walking, running, etc), social interactions (loneliness), mood (bored, focused), computer use, and more.

In analyzing these two popular classes of applications, we identify the following three key design requirements for an inference framework:

*R*1) **Decouple applications, inference algorithms, and devices.** Data driven inferences can often be derived using data from multiple devices. Combining inputs from multiple devices, when available, generally leads to improved inference accuracy (often overlooked by developers [10]). In Figure 1 we illustrate the improvement in inference accuracy for the occupancy and physical activity inferences, used in the Rules and Quantified Self applications respectively; 100% accuracy maps to manually logged ground truth over 28 hours.

Hence, applications should not be restricted to using a single sensor, or a single inference algorithm. At the same time, applications should not be required to incorporate device discovery, handle the challenges of potentially using devices over the wide area (e.g., remote I/O and tolerating disconnections), use disparate device APIs, and instantiate and combine multiple inferences depending on available devices. Therefore an inference framework should require an application to only specify the desired inference, e.g., occupancy (in addition to inference parameters like sampling rate and coverage), while handling the complexity of configuring the right devices and inference algorithms.

*R*2) **Handle environmental dynamics.** Applications are often interested in tracking user and device mobility, and in adapting their processing accordingly. For instance, the QS applications needs to track which locations a user frequents (e.g., home, office, car, gym, meeting room, etc), handle intermittent connectivity, and more. Application development stands to be greatly simplified if the framework were to seamlessly handle such environmental dynamics, e.g., automatically update the selection of devices used for drawing inferences based on user location. Hence the QS application, potentially running on a cloud server, could simply subscribe to the activity inference, which would be dynamically composed of sub-inferences from various devices tracking a user.

*R*3) **Optimize resource usage.** Applications often involve continuous sensing and inferring, and hence consume varying amounts of system resources across multiple devices over time. Such an application must structure its sensing and inference processing across multiple devices, in keeping with the devices' resource constraints. This adds undue burden on developers. For instance, in the QS application, wide area bandwidth constraints may prevent backhaulling of high rate sensor data. Moreover, whenever possible, inferences should be shared across multiple applications to prevent redundant resource consumption. Therefore an inference framework must not only facilitate sharing of inferences, but in doing so must optimize for efficient resource use (e.g., network, battery, CPU, memory, etc) while meeting resource constraints.

Category		**R1**	**R2**	**R3**
Device abstractions	[24] [5] [12] [18]	○		
Mobile sensing	[23] [40]	●		
	[34] [29] [30] [31] [32]	○		○
Cross-device	[49] [45]			●
Macro-programming	[22] [26] [35]	○		

Table 1: Categorization of prior work. ○ denotes partial fulfillment.

3 Related Work

Beam is the first framework that (i) decouples applications, inference algorithms, and devices, (ii) handles environmental dynamics, and (iii) automatically splits sensing and inference logic across devices while optimizing resource usage. We classify prior work into four categories, and compare them based on the requirements above (Table 1).

Device abstraction frameworks: HomeOS [24] and other platforms [5, 12], provide homogeneous programming abstractions to communicate with devices. For instance, HomeOS applications can use a generic motion sensor role, regardless of the sensor's vendor and protocol. Similarly, Rio [18] provides smartphone applications with a uniform device API, regardless of the devices being local or remote. These approaches only decouple device-specific logic from applications, but are unable to decouple inference algorithms from applications. Moreover, they do not address handling of environmental dy-

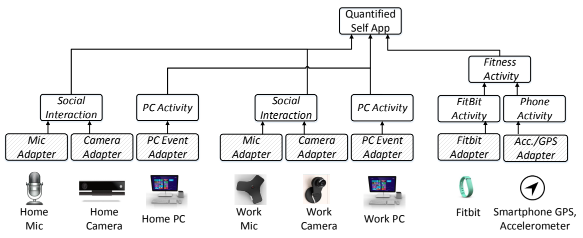

Figure 2: Inference graph for Quantified Self app.

namics and optimizing for efficient resource usage.

Mobile sensing frameworks: Existing work has focused on applications requiring continuous sensing on mobile devices. Kobe [23] and Auditeur [40] propose building libraries of inference algorithms to promote code re-use and to enable developers to select inference algorithms. Other work [29, 30, 31, 32, 34] has focused on improving resource utilization by sharing sensing and data processing across multiple applications on a mobile device. Senergy [32] automates selection of inference algorithms driven by an application requirements on a mobile device. These approaches overlook handling environmental dynamics across devices, and do not address optimizing resource use for inferences across multiple devices. Moreover, they require manual composition of certain inference parameters (e.g., coverage), thus providing limited decoupling of inference algorithms.

Cross-device frameworks: Semantic Streams [49] and Task Cruncher [45] address sharing sensor data and data processing across devices, but are limited to simple stream processing methods, e.g., aggregates, rather than sophisticated inferences. They overlook decoupling of sensing and inferring, as well as handling of dynamics.

Macro-programming frameworks: Macro-programming frameworks [22, 26, 35] provide abstractions to allow applications to dynamically compose dataflows [36, 39]. However these approaches focus on data streaming and aggregations rather than generic inferences, and do not target general purpose compute devices e.g., phones, PCs. In addition, they do not address handling of environmental dynamics and resource optimization across devices at runtime.

4 Beam Inference Framework

We propose Beam as a framework for distributed applications using connected devices. Applications in Beam subscribe to high-level inferences. Beam dynamically composes the required modules to satisfy application requests by using appropriate devices in the given deployment. Central to Beam's design are a set of abstractions that we describe next.

Inference Graphs: Inference graphs are directed acyclic graphs that connect devices to applications. The nodes in this graph correspond to *inference modules* and edges correspond to *channels* that facilitate the transmission of *inference data units (IDUs)* between modules. Figure 2 shows an example inference graph for a Quantified Self application that uses eight different devices spread across the user's home and workplace, and includes mobile and wearable devices.

Composing an inference as a directed graph enables sharing of data processing modules across applications and across modules that require the same input. In Beam, each compute device associated with a user, such as a tablet, phone, PC, or home hub, has a part of the runtime, called the *Engine*. Engines host inference graphs and interface with other engines. Figure 3 shows two engines, one on the user's home hub and another on her phone; the inference graph for QS shown earlier is split across these engines, with the QS application itself running on a cloud server. For simplicity, we do not show another engine that may run on the user's work PC.

IDU: An *Inference data unit (IDU)* is a typed inference, and in its general form is a tuple <t,e,s>, which denotes any inference with state information *s*, generated by an inference algorithm at time *t* and error *e*. The types of the inference state *s*, and error *e*, are specific to the inference at hand. For instance, *s* may be of a numerical type such as a double (e.g., inferred energy consumption), or an enumerated type such as, high, medium, low, or numerical types. Similarly, error *e* may specify a confidence measure (e.g., standard deviation), probability distribution, or error margin (e.g., radius). IDUs abstract away "what is inferred" from "how it is inferred". The latter is handled by inference modules, which we describe next.

Inference Modules: Beam encapsulates inference algorithms into typed modules. Inference modules consume IDUs from one or more modules, perform certain computation using IDU data and pertinent in-memory state, and output IDUs. Special modules called *adapters* interface with underlying sensors and output sensor data as IDUs. Adapters decouple "what is sensed" from "how it is sensed". Module developers specify the IDU types a module consumes, the IDU type it generates, and the module's input dependency (e.g., {PIR} *OR* {camera *AND* mic}). Modules have complete autonomy over how and when to output an IDU, and can maintain arbitrary internal state. For instance, an occupancy inference module may specify (i) input IDUs from microphone, camera, and motion sensor adapters, (ii) allow multiple microphones as input, and (iii) maintain internal state to model ambient noise.

Channels: To ease inference composition, *channels* link modules to each other and to applications. They abstract away the complexities of connecting modules across different devices, including dealing with device disconnec-

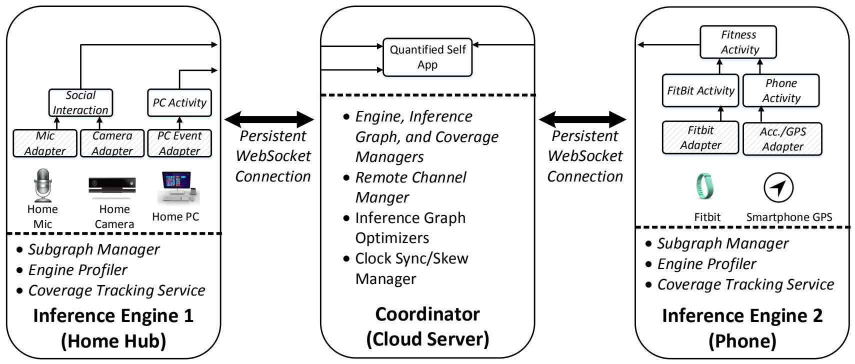

Figure 3: An overview of different components in an example Beam deployment with 2 Engines.

tions, and allowing for optimizations such as batching IDU transfers for efficiency. Every channel has a single *writer* and a single *reader* module. Modules have can have multiple input and output channels. Channels connecting modules on the same engine are *local*. Channels connecting modules on two different engines, across a local or wide area network, are *remote* channels. They enable applications and inference modules to seamlessly use remote devices or remote inference modules.

Coverage tags: Coverage tags help manage sensor coverage. Each adapter is associated with a set of coverage tags which describes what the sensor is sensing. For example, a location string tag can indicate a coverage area such as “home” and a remote monitoring application can use this tag to request an occupancy inference for this coverage area. Coverage tags are strongly typed. Beam uses tag types only to differentiate tags and does not dictate tag semantics. This allows applications complete flexibility in defining new tag types. Adapter are assigned tags by the respective engines at setup time, and are updated at runtime to handle dynamics.

Beam’s runtime also consists of a *Coordinator* which interfaces with all engines in a deployment and runs on a server that is reachable from all engines. The coordinator maintains remote channel buffers to support reader or writer disconnections (typical for mobile devices). It also provides a place to reliably store state of inference graphs at runtime while being resistant to engine crashes and disconnections. The coordinator is also used to maintain reference time across all engines. Engines interface with the coordinator using a persistent web-socket connection, and instantiate and manage the parts of an inference graph(s) local to them.

4.1 Beam Runtime

The Beam runtime creates or updates inference graphs when applications request inferences (R1), and mutates the inference graphs to handle environmental dynamics (R2) and to optimizes resource usage (R3).

Inference graph creation: An application may run on any user device and the sensors required for a requested inference may be spread across devices. Applications request their local Beam engine for all inferences they require. All application requests are forwarded to the coordinator, which uses the requested inference to lookup the required module. It recursively resolves all required inputs of that module (as per its specification), and re-uses matching modules that are already running. The coordinator maintains a *set* of such inference graphs as an *incarnation*. The coordinator determines where each module in the inference graph should run and formulates the new incarnation. The coordinator initializes buffers for remote channels, and partitions the inference graphs into engine-specific subgraphs which are sent to the engines.

Engines receive their respective subgraphs, compare each received subgraph to existing ones, and update them by terminating deleted channels and modules, initializing new ones, and changing channel delivery modes and module sampling rates as needed. Engines ensure that exactly one inference module of each type with a given coverage tag is created.

Inference delivery and guarantees: For each inference request, Beam returns a channel to the application. The inference request consists of i) required inference type or module, ii) *delivery mode*, iii) coverage tags, and iv) sampling requirements (optional).

Delivery mode is a channel property which captures data transport optimizations. For instance, in the *fresh push* mode, an IDU is delivered as soon as the writer-module generates it, while in the *lazy push* mode, the reader chooses to receive IDUs in batches (thus amortizing network transfer costs from battery-limited devices). Remote channels provide IDU delivery in the face of device disconnections by using buffers at the coordinator and the writer engine. Channel readers are guaranteed i) no duplicate IDU delivery, and ii) FIFO delivery based on IDU timestamps. Currently, remote channels use the *drop-tail* policy to minimize wide-area data transfers in the event of a disconnected/lazy reader. This means when a reader re-connects after a long disconnection, it first receives old inference values followed by more recent ones. A *drop-head* policy may be adopted to circumvent this, at the cost of increased data transfers.

In requesting inferences, applications use tags to specify coverage requirements. Furthermore, an application may specify sampling requirements as a latency value that it can tolerate in detecting the change of state for an inference (e.g., *walking* periods of more than 1 minute). This allows adapters and modules to temporarily halt sensing and data processing to reduce battery, network, CPU, or other resources.

Optimizing resource use: The Beam coordinator uses inference graphs as the basis for optimizing resource usage. The coordinator re-configures inference graphs by remapping the engine on which each inference module runs. Optimizations are either performed *reactively* i.e., when an application issues/cancels an inference request, or *proactively* at periodic intervals. Beam's default reactive optimization minimizes the number of remote channels and proactive optimization minimizes the amount of data transferred over remote channels. Other potential optimizations can minimize battery, CPU, and/or memory consumption at engines.

When handling an inference request, the coordinator first incorporates the requested inference graph into the incarnation, re-using already running modules, and merging inference graphs if needed. For new modules, the coordinator decides on which engines they should run (by minimizing the number of remote channels).

Engines profile their subgraphs, and report profiling data (e.g., per-channel data rate) to the coordinator periodically. The coordinator annotates the incarnation using this data and periodically re-evaluates the mapping of inference modules to engines. Beam's default proactive optimization minimizes wide area data transfers.

Handling dynamics: Movement of users and devices can change the set of sensors that satisfy an application requirements. For instance, consider an application that requires camera input from the device currently facing the user at any time, such as the camera on her home PC, office PC, smartphone, etc. In such scenarios, the inference graph needs to be updated dynamically. Beam updates the coverage tags to handle such dynamics. Certain tags such as those of *location* type (e.g., "home") can be assumed to be static (edited only by the user), while for certain other types, e.g, *user*, the sensed subject is mobile and hence the sensors that cover it may change.

The coordinator's *tracking service* manages the coverage tags associated with adapters on various engines. The engine's tracking service updates the user coverage tags as the user moves. For example, when the user leaves her office and arrives at home, the tracking service removes the *user* tag from device adapters in the office, and adds them to adapters of devices deployed in the home. The tracking service relies on device interactions to track users. When a user interacts with a device it updates the tags of all sensors on the device to include the user's tag.

When coverage tags change (e.g., due to user movement and change in sensor coverage), the coordinator recomputes the inference graphs and sends updated subgraphs to the affected engines.

4.2 Implementation

Our Beam prototype is implemented in C# as a cross-platform portable service that can be used by .NET v4.5, Windows Store 8.1, and Windows Phone 8.1 applications. Module binaries are currently wrapped within the service, but may also be downloaded from the coordinator on demand. We have implemented 8 inference modules: mic-occupancy [27], camera-occupancy, appliance-use [21, 28], occupancy, pc-activity [37], fitness-activity [43], semantic-location, social-interaction, and 9 adapters: tablet and pc mic, power-meter, fitbit [4], GPS, accelerometer, pc-interaction, pc-event, and a HomeOS [24] adapter to access all its device drivers. We have implemented the two sample applications, described in Sec 2. Both applications run on a cloud VM; Beam hosts the modules for their inferences across the user's home PC, work PC, and phone.

Compared to monolithic or library based approaches, we find that for these applications using Beam's framework results in up to 4.5× lower number of tasks and 12× lower SLoC, up to 3× higher inference accuracy due to Beam's handling of environmental dynamics, and Beam's dynamic optimizations match hand optimized versions for network resource usage. We aim to enrich Beam's optimizer to include optimizations for battery, CPU, and memory, and plan to extend tracking support to objects using passive tags, e.g., RFID, or QR codes [17].

5 Conclusion

Applications using connected devices are difficult to develop today because they are constructed as monolithic silos, tightly coupled to sensing devices, and must implement all data sensing and inference logic, even as devices move or are temporarily disconnected. We present *Beam*, a framework and runtime for distributed inference-driven applications that (i) decouples applications, inference algorithms, and devices, (ii) handles environmental dynamics, and (iii) automatically splits sensing and inference logic across devices while optimizing resource usage. Using Beam, applications only specify "what should be sensed or inferred," without worrying about "how it is sensed or inferred." Beam simplifies application development and maximizes the utility of user-owned devices. *It is time to end monolithic apps for connected devices.*

References

[1] Amazon Home Automation Store. http://www.amazon.com/home-automation-smarthome/b?ie=UTF8&node=6563140011.

[2] The US Market for Home Automation and Security Technologies. Technical report, BCC Research, IAS031B, 2011.

[3] Dropcam - Super Simple Video Monitoring and Security. https://www.dropcam.com/.

[4] Fitbit. https://www.fitbit.com/.

[5] HomeSeer. http://homeseer.com/.

[6] iOS: Understanding iBeacon. http://support.apple.com/kb/HT6048.

[7] IFTTT: Put the internet to work for you. https://ifttt.com/.

[8] The Internet of Things. http://share.cisco.com/internet-of-things.html/.

[9] Map my fitness. http://www.mapmyfitness.com/.

[10] Nest. http://www.nest.com/.

[11] Quantified self. http://quantifiedself.com/.

[12] Revolv. http://revolv.com/.

[13] ShopKick shopping app. http://shopkick.com/.

[14] Simplicam Home Surveillance Camera. https://www.simplicam.com/.

[15] Smart Power Strip, . http://www.rogeryiu.com/.

[16] SmartThings, . http://www.smartthings.com/.

[17] https://www.thetileapp.com/.

[18] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A System Solution for Sharing I/O between Mobile Systems. In *Proc. ACM MobiSys*, 2014.

[19] E. Arroyo, L. Bonanni, and T. Selker. Waterbot: Exploring feedback and persuasive techniques at the sink. In *Proc. ACM CHI 2005*.

[20] R. K. Balan, K. X. Nguyen, and L. Jiang. Real-time trip information service for a large taxi fleet. In *Proc. 9th ACM MobiSys*, June 2011.

[21] N. Batra, J. Kelly, O. Parson, H. Dutta, W. J. Knottenbelt, A. Rogers, A. Singh, and M. Srivastava. Nilmtk: an open source toolkit for non-intrusive load monitoring. In *Proc. ACM e-Energy*, 2014.

[22] A. Boulis, C.-C. Han, R. Shea, and M. B. Srivastava. Sensorware: Programming sensor networks beyond code update and querying. *Pervasive Mob. Comput.*, 2007.

[23] 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. In *Proc. 9th ACM SenSys*, Nov. 2011.

[24] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Proc. 9th USENIX NSDI*, Apr. 2012.

[25] J. Froehlich, L. Findlater, M. Ostergren, S. Ramanathan, J. Peterson, I. Wragg, E. Larson, F. Fu, M. Bai, S. Patel, and J. A. Landay. The design and evaluation of prototype eco-feedback displays for fixture-level water usage data. In *ACM CHI 2012*.

[26] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *Distributed Computing in Sensor Systems*. 2005.

[27] T. Hao, G. Xing, and G. Zhou. isleep: Unobtrusive sleep quality monitoring using smartphones. In *Proc 11th ACM SenSys*, 2013.

[28] G. Hart. Nonintrusive appliance load monitoring. *Proceedings of the IEEE*, 1992.

[29] 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 *Proc. 10th ACM SenSys*, Nov. 2012.

[30] 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. In *Proc. ACM MobiSys*, 2008.

[31] S. Kang, Y. Lee, C. Min, Y. Ju, T. Park, J. Lee, Y. Rhee, and J. Song. Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *IEEE PerCom*, 2010.

[32] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *Proc. ACM OOPSLA*, Nov. 2013.

[33] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the north sea. In *Proc. 3rd ACM SenSys*, Nov. 2005.

[34] F. Lai, S. S. Hasan, A. Laugesen, and O. Chipara. Csense: A stream-processing toolkit for robust and high-rate mobile sensing applications. In *Proc. 13th*

IPSN, 2014.

[35] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 2006.

[36] G. Mainland, M. Welsh, and G. Morrisett. Flask: A language for data-driven sensor network programs. *Harvard Univ., Tech. Rep. TR-13-06*, 2006.

[37] G. Mark, S. T. Iqbal, M. Czerwinski, and P. Johns. Bored mondays and focused afternoons: The rhythm of attention and online activity in the workplace. In *Proc. of the 32nd ACM CHI*, April 2014.

[38] D. Morris, A. B. Brush, and B. R. Meyers. Superbreak: Using interactivity to enhance ergonomic typing breaks. In *Proc. ACM CHI*, 2008.

[39] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Proc. of the 6th ACM/IEEE IPSN*, 2007.

[40] 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 *Proc. 11th ACM MobiSys*, June 2013.

[41] M. M. Rahman, A. A. Ali, K. Plarre, M. al'Absi, E. Ertin, and S. Kumar. mConverse: Inferring conversation episodes from respiratory measurements collected in the field. In *Proceedings of the 2nd Conference on Wireless Health*, 2011.

[42] H. Ramamurthy, B. S. Prabhu, R. Gadh, and A. Madni. Wireless industrial monitoring and control using a smart sensor platform. *Sensors Journal, IEEE*, 2007.

[43] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Using mobile phones to determine transportation modes. *ACM Transactions on Sensor Networks (TOSN)*, 2010.

[44] R. P. Singh, S. Keshav, and T. Brecht. A cloud-based consumer-centric architecture for energy data analytics. In *e-Energy*, 2013.

[45] A. Tavakoli, A. Kansal, and S. Nath. On-line sensing task optimization for shared sensors. In *Proc. 9th ACM/IEEE IPSN*, 2010.

[46] T. Toscos, A. Faber, S. An, and M. P. Gandhi. Chick clique: Persuasive technology to motivate teenage girls to exercise. In *ACM CHI EA 2006*.

[47] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman. Practical trigger-action programming in the smart home. In *Proc. ACM CHI*, 2014.

[48] R. Wang, F. Chen, Z. Chen, T. Li, G. Harari, S. Tignor, X. Zhou, D. Ben-Zeev, and A. T. Campbell. Studentlife: Assessing behavioral trends, mental well-being and academic performance of college students using smartphones. In *Proc. ACM UbiComp*, 2014.

[49] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *In Proceedings of the European Workshop on Wireless Sensor Networks*, 2006.

[50] D. Wyatt, T. Choudhury, J. Bilmes, and J. A. Kitts. Inferring colocation and conversation networks from privacy-sensitive audio with implications for computational social science. *ACM Trans. Intell. Syst. Technol.*, 2011.