Pixie: An Operating System for Resource-Aware Programming of Embedded Sensors

Konrad Lorincz, Bor-rong Chen, Jason Waterman,
Geoffrey Werner-Allen, and Matt Welsh
School of Engineering and Applied Sciences, Harvard University
{konrad,brchen,waterman,werner,mdw}@eecs.harvard.edu

Abstract

A growing class of sensor network applications require high data rates and computationally-intensive node-level processing. When deployed into environments where resources are limited and variable, achieving good performance requires applications to adjust their behavior as resource availability changes. This paper presents *Pixie*, a new sensor network operating system designed to facilitate the design of highly-efficient resource-aware applications. By allowing applications to introspect on resource availability and providing a rich interface for controlling resource usage, Pixie enables a broad range of adaptation policies through a small set of core abstractions.

Categories and Subject Descriptors

D.4.7 [Operating Systems]: Organization and Design; D.1 [Programming Techniques]; C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms

Design

Keywords

Resource-Aware Programming, Resource Reservations, Wireless Sensor Networks

1 Introduction

Sensor networks are becoming increasingly important for data-intensive applications that involve moderate to high data rates, fine-grained timestamping of recorded signals, and computationally-intensive processing. Examples of such applications include seismic monitoring of earthquake zones and volcanoes [10], structural health monitoring [11], and biomedical data capture using tiny, wearable sensors [7]. In contrast to the first generation of sensor networks, which were focused on low-duty-cycle data collection and aggregation, these new applications demand much greater data fidelity and computational sophistication.

At the same time, wireless sensor platforms are inherently resource-constrained in order to achieve low power consumption. This leads to severe limitations of computational horsepower, memory capacity, and radio bandwidth. The stringent application demands and resource constraints require that applications be *resource-aware* to realize efficient implementations.

In this paper we present *Pixie*, a new sensor node operating system designed to overcome the resource-management challenges facing sensor application developers. Pixie is focused on enabling the development of *resource-aware* applications that receive feedback on and react to changes in resource availability at runtime. We argue that resource awareness is a fundamental requirement for sensor application development, especially in contexts where the processing overheads, memory consumption, and bandwidth usage exhibit wide variations during the network's lifetime. Pixie makes the following contributions:

- A dataflow-oriented programming model that enables a resource aware programming style, providing visibility and control over resource usage to applications;
- A fine-grained approach to resource reservation and accounting, based on a small set of core abstractions; and
- A system architecture that supports interchangeable and composable policies for resource management.

As an example, we consider a sensor network for monitoring limb motion in a patient with Parkinson's disease. This system will experience fluctuations in radio bandwidth (due to mobility), CPU demand (due to the nature of the recorded data), and energy availability (due to the

HotEmNets'08, June 2-3, 2008, Charlottesville, Virginia, USA

limited battery charge). It is not generally possible to statically allocate sensor node resources to meet these varying demands; rather, the application must take an active role in prioritizing which signals to store, process, and transmit depending on current resource availability.

The Pixie architecture makes use of a staged concurrency model [9] in which application logic is partitioned into a dataflow graph of stages connected by request queues. This model allows the OS and application to negotiate for resource allocations at a stage level, and makes resource bottlenecks and request streams visible to the OS. Likewise, Pixie provides stages with feedback on available resources. This allows application logic to make informed decisions about resource constraints, such as decimating an input signal or degrading the amount of processing performed in real time.

In this paper we describe the overall Pixie architecture, present a motivating application, and show preliminary results demonstrating the system's ability to support resource-aware programming.

2 Motivation and Background

A growing number of sensor network applications require moderate to high data rates, high data fidelity, and computationally-intensive processing. As resources in this environment are limited and highly dynamic, applications must be aware of the runtime resource conditions and be able to tune their behavior accordingly. To motivate our architecture, we present an example application requiring adaptation to bandwidth and energy availability. Our group is investigating the use of wireless, wearable sensors, capturing high-resolution data on limb motion and muscle use to aid in the treatment of patients with Parkinson's disease. Studies involve a patient wearing up to 10 tiny, wireless sensors on their limbs and torso, each consisting of a triaxial accelerometer and gyroscope sampled at 100 Hz allowing the patient's activity level and specific motion disorders to be characterized.

This system collects much more data than can be transmitted over low bandwidth sensor radios, and the energy consumed by transmitting raw data has a substantial impact on node lifetime. One way to save bandwidth is to compute high-level features from the raw signals on the nodes, such as the dominant frequency, peak amplitude, and RMS, which are ultimately used for classifying patient activity. Likewise, a simple "stillness filter" can disable processing and transmissions altogether when a given sensor node is not moving. Estimating the amount of bandwidth available for data transmission requires estimating radio link conditions, which can be affected by patient location, interference, and other factors. In the extreme case, the sensor network can experience periods of disconnection from the base station (due to patient mobility), which force data to be queued up for later delivery.

In this application, bandwidth and energy availability cannot be predicted *a priori*. Therefore, achieving good performance requires the application to adapt its behavior when resource availability changes, for example, by varying the type of features computed by the sensor node as well as the type and amount of data transmitted. Making these adjustments properly requires weighing the relative utility of each operation to the application against the corresponding cost in energy and bandwidth.

2.1 Related Work

Currently, achieving good resource efficiency from these kinds of applications is still a tremendous challenge, and current programming models do little to ease the burden. Sensor network operating systems such as TinyOS [4], Contiki [3], and MantisOS [1] provide resource management only through low-level primitives for manipulating hardware states. These systems require that applications perform their own scheduling, bandwidth estimation, and power management.

ECOSystem [12] and Odyssey [6] represent two approaches to managing resource adaptivity in mobile systems. While Odyssey provides callbacks to applications, permitting them to respond to varying energy, bandwidth, and computational load, ECOsystem does not require application changes, automatically adjusting CPU scheduling parameters to manage energy consumption. Because both ECOSystem and Odyssey interface to applications through the standard UNIX interfaces, they lack the knowledge of the application structure our data-flow model elicits and the control it exposes. Also, these systems are focused on very different hardware platforms and application requirements than those found in sensor networks.

Eon [8] is one of the first sensor network platforms making resource-awareness a first-class programming model primitive. Eon adapts application energy consumption to availability by tuning dataflow paths and timer rates in the application. Programmer control is limited to providing multiple data paths with varying resource consumption, and no feedback on resource availability is delivered to the application. Eon does not address the more general issue of adapting to CPU load, memory pressure, or bandwidth fluctuations.

A significant difference between Eon and Pixie is that Eon's adaptation techniques effectively couple energy-availability and output data quality in a way we believe is limiting for sensor network applications. For example, in the motion analysis case study discussed previously, Eon would send all data down the high-fidelity path when power was available, and all data down the low-fidelity path when power was scarce, regardless of the corresponding data utility. This can lead to inefficiency, since resource-management decisions are not data-dependent. Other systems, such as EnviroMic [5], adapt resource consumption to input data but do so without considering available energy. In contrast, Pixie is designed to allow applications more control over where energy is used at all resource availability levels. This allows applications to make decisions incorporating both the value of sampled data to the end-user and available resources.

The core challenge that we face is how to simultaneously enable resource-aware applications while min-

imizing programmer burden. Exposing a vast array of low-level "knobs" gives programmers the most control, but does little to simplify application design. On the other hand, giving the OS sole responsibility for resource management leads to inefficient use of already scarce resources. In Pixie, our goal is to strike a balance between these extremes and achieve near optimal resource usage while greatly easing the process of writing resource-aware code.

3 Pixie Architecture

The Pixie architecture consists of three components. The Application Dataflow layer implements application logic, the Resource Manager allocates and manages resources for the application, and the Scheduler schedules and executes application stages. In the remainder of this section we present the core architecture by describing each of the three components in detail. To further illustrate the Pixie architecture, we present concrete examples from our motion analysis application.

3.1 Application Dataflow

Pixie applications are composed as a dataflow graph of stages. Stages process sensor data or perform low-level operations such as sampling, radio communication, or reading and writing to flash. Stages are connected by edges that are connected to each stage's input and output ports. By default, each stage has a single input port and single output port, although multiple ports may be used. Edges themselves perform no queuing; a special queue stage can be introduced into the stage graph if needed.

We settled on a dataflow programming model for several reasons. First, dataflow maps well to typical sensor network application design and provides greater structure than an arbitrary graph of software components (a la NesC). Second, it provides a uniform interface between application modules, which allows the OS to reason about the application at a fairly fine level of granularity. The dataflow model gives the OS both visibility and control at a stage level over resource allocations, scheduling, and the flow of data through the application. Third, dataflow graphs naturally support interpositioning, which is a core mechanism used by Pixie's resource manager to affect control over the application's resource usage.

Figure 1 shows a simplified version of the motion analysis application running on top of Pixie. A sampling component (parametrized by a sampling rate) feeds raw sensor data into an activity detector, which determines whether the signal contains motion; if not, the signal is discarded. Otherwise raw data is passed to a series of stages performing feature extraction, such as computing peak amplitude (PA), root-mean-square (RMS), and signal decimation. In addition, raw signal data is written to flash memory for later retrieval. Features are then delivered to a radio component for transmission to the base station.

Application stages require resources to process data from their input queue. We assume that each stage knows

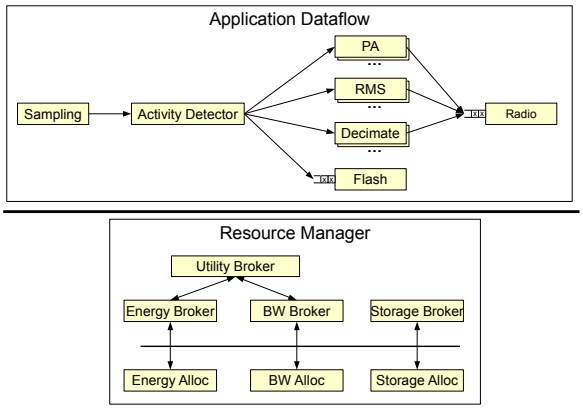

Figure 1: **Simplified motion analysis architecture.**

the type and amount of resources it needs in order to process one item from its input queue. In general, a stage requires a resource vector to complete its task. A *resource vector* consists of a list of resource types and quantities. For example, sending data over the radio requires both energy and bandwidth resource units. A stage makes an asynchronous resource request to Pixie specifying its resource vector. At a later time Pixie replies to the request with either a granted or declined response. If the reply is granted, the stage becomes runnable and can be scheduled for execution.

3.2 Resource Manager

One of the primary goals of Pixie is to manage resources at a fine granularity level. In our dataflow model, stages can perform a wide variety of tasks with different resource requirements. To facilitate resource management at the stage level, the Pixie architecture provides the mechanism in the form of tickets and allocators, and the ability to express policies through brokers.

3.2.1 Tickets and allocators

A *ticket* represents a right to consume a given amount of a resource type within a time window. A ticket is a tuple of three values: resource type, quantity, and expiration time. A ticket is allocated to a stage if the request is granted, at which point the ticket is outstanding. A stage redeems a ticket prior to using the associated resource; upon use, the ticket is consumed. A stage may relinquish a ticket if it has no further use for it. Finally, a ticket may be revoked at any time, for example, if the resource is no longer available.

Tickets have an expiration time for two reasons. First, resource availability estimates might be accurate only over a limited time horizon (as in the case of bandwidth estimation). Second, ticket expiry makes accountability easier to manage, since it bounds the number of outstanding tickets. Furthermore, expiry prevents resource leakage (analogous to a garbage collection in programming languages).

When a ticket is redeemed, if only a fraction of the

ticket's resource quantity is used, the remainder is returned to the system rather than the application. This simplifies application design and reduces overhead because it relieves the application from having to deal with tickets containing small resource quantities. However, we do provide a mechanism for the application to split a ticket into two or more tickets when explicitly needed.

An *allocator* estimates the available quantity of a resource type and provides the mechanism for allocating and tracking tickets. Allocators simply allocate tickets upon request as long as the requested resources are available. It is up to applications (and brokers, as described below) to implement policies for resource management on top of the allocator.

As an example, consider the energy allocator from our motion analysis application in Figure 1. The energy allocator keeps track of available battery charge and allocates requests as long as the battery is not completely drained. We do not use explicit ticket allocation for low power rudimentary operations and treat them as static constant energy drain. However, we do explicitly allocate energy for stages consuming substantially more than nominal amounts, such as transmitting a radio packet, writing or reading from flash, changing the CPU frequency (e.g., on the iMote2), or performing a significant amount of computation.

Another example is a bandwidth allocator that estimates bandwidth availability over a short time window (e.g. 10 sec) and allocates tickets based on this estimate. Because bandwidth availability may change even more frequently than the estimation window (in the extreme case loosing connectivity), outgoing packets are queued for transmission. The bandwidth allocator's goal is to provide a low queuing delay and minimize the number of ticket revocations.

Finally, a flash storage allocator tracks the amount of free storage and allocates flash storage grants. As with energy and bandwidth, if there is available free storage, the request is granted. A similar allocation mechanism is used for memory.

3.2.2 Brokers

Tickets and allocators provide a low-level interface for resource management. This design permits a wide range of policies to be layered over this interface, although programming at the ticket level can be difficult. In order to simplify application design, we propose a *resource broker* abstraction that implements resource management policies on behalf of the application. A broker acts as an intermediary between stages and allocators. Brokers are entirely optional, however we anticipate that most applications will want to take advantage of them. Pixie provides a standard library of brokers from which an application may choose.

For example, Pixie's energy broker implements a policy for the system to achieve a target lifetime. In this case, the energy broker will trickle the amount of energy the application can use at a specific rate. Another broker may implement a greedy policy for bandwidth usage in which it allows any packet to be transmitted as long as there is available bandwidth. Another bandwidth broker may try to eliminate bursty transmissions by implementing a leaky bucket approach. An even more sophisticated bandwidth broker will negotiate bandwidth usage with neighboring nodes in order to achieve fairness.

Brokers may be stacked or layered in various ways. For example, in Figure 1 brokers are composed into a hierarchy. The first level implements policies pertaining to a single resource, such as for bandwidth, energy, and flash storage. The second level implements an application specific utility broker. In our application different data features provide various degrees of information about the signal and therefore are assigned different utility values. For example, peak amplitude features provide more useful information about the signal than RMS features, and are therefore assigned a higher utility. Likewise, RMS features are assigned a higher utility than decimated samples. When assigning resources to the feature stages, the utility broker performs a first fit bin packing ordered by stage priority.

3.3 Scheduler

The scheduler is responsible for scheduling and running stages. Due to memory limitations on typical sensor node platforms, we assume a single thread of execution and a non-preemptive scheduling policy, avoiding the need to maintain potentially large stacks across context switches.

In Pixie, scheduling decisions are made based on the scheduling policy used and if a stage is blocking. For example, a stage may block if it was not granted the requested resources. At a later time, the stage may become unblocked by receiving a signal from the resource manager with the requested resource grant.

Pixie admits a wide range of CPU scheduling policies. Our default implementation performs a simple depth-first traversal of the stage graph, directly invoking downstream stages using a procedure call, thereby minimizing cross-stage overhead. Traversals are initiated at *source stages*, such as sampling or timers, and terminate at *sink stages* which either do not push data to an output port, or have no output ports. A *queue stage* acts as both a sink stage and a source stage; the queue initiates graph traversals downstream when the stage is non-empty. Each source stage has an associated *priority*, which may be altered at runtime. The scheduler initiates a graph traversal at the source stage with the highest priority, and ties are broken using round-robin.

4 Preliminary Results

This section describes a preliminary implementation of Pixie in the context of a simplified version of our motion analysis application presented in Section 3. Due to lack of space, we focus on presenting initial results demonstrating Pixie's effectiveness in enabling resource awareness, by allowing the application to adapt to varying radio bandwidth availability.

We have implemented a prototype of Pixie in NesC

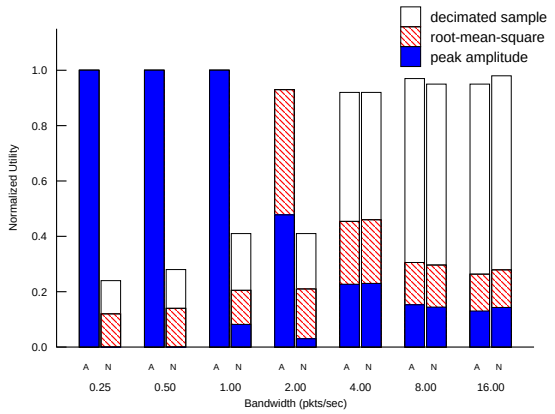

Figure 2: **Normalized data utility achieved by an adaptive (A) and a non-adaptive (N) motion analysis applications.**

on the SHIMMER [7] sensor node platform. SHIMMER has a TI MSP430 microcontroller, CC2420 802.15.4 radio, 2 GB microSD flash, triaxial accelerometer and gyroscope sensors. Each node samples six channels of sensor data at 100 Hz and computes peak amplitude and RMS features. Nodes also transmit a 20 Hz decimation of the raw signal to aid calibration and visualization.

When bandwidth is constrained, the application's goal is to prioritize transmission of the most important data to the base station. We implement a bandwidth allocator that periodically estimates available bandwidth using the expected transmission count (ETX) metric [2]. To express the value of data types, we assign each data product a utility score: 20 for peak amplitude, 10 for RMS and 1 for decimated samples. We use the utility broker described in Section 3.2.2 to allocate bandwidth to stages.

In order to evaluate the effectiveness of bandwidth management, we compare adaptive and non-adaptive versions of the motion analysis application. The two versions of the application are identical except that the non-adaptive version does not use a utility broker to prioritize transmissions. Instead, transmissions are scheduled in a round robin fashion. To facilitate experimentation, we artificially constrain available bandwidth at each node. Pixie's bandwidth allocator measures the altered capacity using the ETX metric.

As our comparison metric, we define aggregate utility as the sum of the utility values for all data received by the base station. Figure 2 shows the aggregate utility, normalized to the maximum achievable value under each bandwidth capacity setting. As the figure shows, the two versions are equivalent when there is sufficient bandwidth to send every packet. However, when bandwidth is constrained, the adaptive version transmits only high utility data while the non-adaptive version does not distinguish between data types. This behavior is straightforward to achieve with Pixie by inserting a data-aware bandwidth broker into the application.

5 Conclusions

We are driving Pixie's design through close collaboration with domain scientists, who demand an easier approach to resource management than afforded by existing systems. We argue that resource management is *the* key barrier to adoption of wireless sensor networks in many applications, since constrained resources underpin nearly all aspects of the programming model, including concurrency, memory management, and radio communication. Further, we claim that no OS is completely capable of managing resources transparently to the application: the application must be involved in the process of adapting to changing conditions, either through direct resource awareness, or by specifying policies that enable the system to make the right decisions. Although our results are preliminary, we have demonstrated that Pixie provides an effective platform for adapting to varying radio bandwidth. Based on our early experience, we believe that the Pixie design also supports a range of policies for managing CPU, memory, and energy for adaptive sensor network applications.

References

[1] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *ACM/Kluwer Mobile Networks and Applications (MONET), Special Issue on Wireless Sensor Networks*, 10(4):563–579, August 2005.

[2] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *MOBICOM '03*, San Diego, California, September 2003.

[3] A. Dunkels, B. Grnvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Emnets-I*, Tampa, Florida, USA, Nov. 2004.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *ASPLOS*, Boston, MA, USA, Nov. 2000.

[5] L. Luo, Q. Cao, C. Huang, T. Abdelzaher, J. A. Stankovic, , and M. Ward. Enviromic: Towards cooperative storage and retrieval in audio sensor networks. In *ICDCS*, June 2007.

[6] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *SOSP*, 1997.

[7] S. Patel, K. Lorincz, R. Hughes, N. Huggins, J. H. Growdon, M. Welsh, and P. Bonato. Analysis of feature space for monitoring persons with parkinson's disease with application to a wireless wearable sensor system. In *29th IEEE EMBS Conference*, August 2007.

[8] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. Corner, and E. Berger. Eon: A language and runtime system for perpetual systems. In *Proc. Fifth ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[9] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. the 18th SOSP*, Banff, Canada, October 2001.

[10] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. the 7th OSDI*, Seattle, Washington, November 2006.

[11] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proc. Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[12] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: managing energy as a first class operating system resource. In *ASPLOS-X*, New York, NY, USA, 2002.