System Architecture Directions for Networked Sensors *

Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA

{jhill, szewczyk, awoo, culler}@cs.berkeley.edu, {shollar, pister}@eecs.berkeley.edu

ABSTRACT

Technological progress in integrated, low-power, CMOS communication devices and sensors makes a rich design space of networked sensors viable. They can be deeply embedded in the physical world and spread throughout our environment like smart dust. The missing elements are an overall system architecture and a methodology for systematic advance. To this end, we identify key requirements, develop a small device that is representative of the class, design a tiny event-driven operating system, and show that it provides support for efficient modularity and concurrency-intensive operation. Our operating system fits in 178 bytes of memory, propagates events in the time it takes to copy 1.25 bytes of memory, context switches in the time it takes to copy 6 bytes of memory and supports two level scheduling. The analysis lays a groundwork for future architectural advances.

1. INTRODUCTION

As the post-PC era emerges, several new niches of computer system design are taking shape with characteristics that are quite different from traditional desktop and server regimes. Many new regimes have been enabled, in part, by "Moore's Law" pushing a given level of functionality into a smaller, cheaper, lower-power unit. In addition, three other trends are equally important: complete systems on a chip, integrated low-power communication, and integrated low-power transducers. All four of these trends are working together to enable the networked sensor. The basic microcontroller building block now includes not just memory and processing, but non-volatile memory and interface resources, such as DACs, ADCs, UARTs, interrupt controllers, and counters. Communication can now take the form of wired, short-range RF, infrared, optical, and various other techniques [18]. Sensors now interact with various fields and forces to detect light, heat, position, movement, chemical presence, and so on. In each of these areas, the technology is crossing a critical threshold that makes networked sensors an exciting regime to apply systematic design methods.

Today, networked sensors can be constructed using commercial components on the scale of a square inch in size and a fraction of a watt in power. They use one or more microcontrollers connected to various sensor devices and to small transceiver chips. One such sensor is described in this study. Many researchers envision driving the networked sensor down to microscopic scale by taking advantage of advances in semiconductor processes. This includes having communication integrated on-chip with a rich set of microelectromechanical (MEMS) sensors and CMOS logic at extremely low cost [37, 5]. They envision that this **smart dust** will be integrated into the physical environment, perhaps even powered by ambient energy [31], and used in many smart space scenarios. Alternatively, others envision ramping up the functionality associated with one-inch devices dramatically. In either scenario, it is essential that the network sensor design regime be subjected to the same rigorous, workload-driven, quantitative analysis that allowed microprocessor performance to advance so significantly over the past 15 years. It should not be surprising that the unique characteristics of this regime give rise to very different design trade-offs than current general-purpose systems.

This paper provides an initial exploration of system architectures for networked sensors. The investigation is grounded in a prototype "current generation" device constructed from off-the-shelf components. Other research projects [37, 5] are trying to compress this class of devices onto a single chip. The key missing technology is the system software support to manage and operate the device. To address this problem, we have developed a tiny microthreaded OS, called TinyOS. It draws on previous architectural work on lightweight thread support and efficient network interfaces. While working in this design regime two issues emerge strongly: these devices are *concurrency intensive* - several different flows of data must be kept moving simultaneously; and the system must provide *efficient modularity* - hardware specific and application specific components must snap together with little processing and storage overhead. We address these two problems with our tiny microthreaded OS. Analysis of this solution provides valuable initial directions for future architectural innovation.

*This work is supported, in part, by the Defense Advanced Research Projects Agency (grant DABT 63-98-C-0038, "Ninja", and contract DABT63-98-1-0018, "Smart Dust") and the National Science Foundation (grant RI EIA-9802069). Support is provided as well by Intel Corporation, Ericsson, Philips, Sun Microsystems, IBM, Nortel Networks, and Compaq.

ASPLOS 2000 **Cambridge, MA** **Nov. 12-15 , 2000**

Section 2 outlines the design requirements that characterize the networked sensor regime and guide our microthreading approach. Section 3 describes our baseline, current-technology hardware design. Section 4 develops our TinyOS for devices of this general class. Section 5 evaluates the effectiveness of the design against a collection of preliminary benchmarks. Section 6 contrasts our approach with that of prevailing embedded operating systems. Finally, Section 7 draws together the study and considers its implications for architectural directions.

2. NETWORKED SENSOR CHARACTERISTICS

This section outlines the requirements that shape the design of network sensor systems; these observations are made more concrete by later sections.

Small physical size and low power consumption: At any point in technological evolution, size and power constrain the processing, storage, and interconnect capability of the basic device. Obviously, reducing the size and power required for a given capability are driving factors in the hardware design. Likewise, the software must make efficient use of processor and memory while enabling low power communication.

Concurrency-intensive operation: The primary mode of operation for these devices is to flow information from place to place with a modest amount of processing on-the-fly, rather than to accept a command, stop, think, and respond. For example, information may be simultaneously captured from sensors, manipulated, and streamed onto a network. Alternatively, data may be received from other nodes and forwarded in multi-hop routing or bridging situations. There is little internal storage capacity, so buffering large amounts of data between the inbound and the outbound flows is unattractive. Moreover, each of the flows generally involve a large number of low-level events interleaved with higher-level processing. Some of the high-level processing will extend over multiple real-time events.

Limited Physical Parallelism and Controller Hierarchy: The number of independent controllers, the capabilities of the controllers, and the sophistication of the processor-memory-switch level interconnect are much lower than in conventional systems. Typically, the sensor or actuator provides a primitive interface directly to a single-chip microcontroller. In contrast, conventional systems distribute the concurrent processing associated with the collection of devices over multiple levels of controllers interconnected by an elaborate bus structure. Space and power constraints and limited physical configurability on-chip are likely to drive the need to support concurrency-intensive management of flows through the embedded microprocessor.

Diversity in Design and Usage: Networked sensor devices will tend to be application specific, rather than general purpose, and carry only the available hardware support actually needed for the application. As there is a wide range of potential applications, the variation in physical devices is likely to be large. On any particular device, it is important to easily assemble just the software components required to synthesize the application from the hardware components. Thus, these devices require an unusual degree of software *modularity* that must also be very efficient. A generic development environment is needed which allows specialized applications to be constructed from a spectrum of devices without heavyweight interfaces. Moreover, it should be natural to migrate components across the hardware/software boundary as technology evolves.

Robust Operation: These devices will be numerous, largely unattended, and expected to form an application which will be operational a large percentage of the time. The application of traditional redundancy techniques to enhance the reliability of individual units is limited by space and power. Although redundancy across devices is more attractive than within devices, the communication cost for cross device failover is prohibitive. Thus enhancing the reliability of individual devices is essential. Additionally, we can increase the reliability of the *application* by tolerating individual device failures. To that end, the operating system running on a single node should not only be robust, but also should facilitate the development of reliable distributed applications.

3. EXAMPLE DESIGN POINT

To ground our system design study, we have developed a small, flexible networked sensor platform that has many of the key characteristics of the general class and utilizes the various internal interfaces using currently available components [33]. A photograph and schematic for the hardware configuration of this device appear in Figure 1. It consists of a microcontroller with internal flash program memory, data SRAM and data EEPROM, connected to a set of actuator and sensor devices, including LEDs, a low-power radio transceiver, an analog photo-sensor, a digital temperature sensor, a serial port, and a small coprocessor unit. While not a breakthrough in its own right, this prototype has been invaluable in developing a feel for the salient issues in this design regime.

3.1 Hardware Organization

The processor within the MCU (ATMEL 90LS8535) [2], which conventionally receives so much attention, is not particularly noteworthy. It is an 8-bit Harvard architecture with 16-bit addresses. It provides 32 8-bit general registers and runs at 4 MHz and 3.0 V. The system is very memory constrained: it has 8 KB of flash as the program memory, and 512 bytes of SRAM as the data memory. The MCU is designed such that the processor cannot write to instruction memory; our prototype uses a coprocessor to perform that function. Additionally, the processor integrates a set of timers and counters which can be configured to generate interrupts at regular time intervals. More noteworthy are the three sleep modes: *idle*, which just shuts off the processor, *power down*, which shuts off everything but the watchdog and asynchronous interrupt logic necessary for wake up, and *power save*, which is similar to the power down mode, but leaves an asynchronous timer running.

Three LEDs represent outputs connected through general I/O ports; they may be used to display digital values or status. The photo-sensor represents an analog input device with simple control lines. In this case, the control lines eliminate power drain through the photo resistor when not in use. The input signal can be directed to an internal ADC in continuous or sampled modes.

The radio is the most important component. It represents an asynchronous input/output device with hard real time constraints. It consists of an RF Monolithics 916.50

MHz transceiver (TR1000) [10], antenna, and collection of discrete components to configure the physical layer characteristics such as signal strength and sensitivity. It operates in an ON-OFF key mode at speeds up to 19.2 Kbps. Control signals configure the radio to operate in either transmit, receive, or power-off mode. The radio contains no buffering so each bit must be serviced by the controller on time. Additionally, the transmitted value is not latched by the radio, so jitter at the radio input is propagated into the transmission signal.

The temperature sensor (Analog Devices AD7418) represents a large class of digital sensors which have internal A/D converters and interface over a standard chip-to-chip protocol. In this case, the synchronous, two-wire I^2C [39] protocol is used with software on the microcontroller synthesizing the I^2C master over general I/O pins. In general, up to eight different I^2C devices can be attached to this serial bus, each with a unique ID. The protocol is rather different from conventional bus protocols, as there is no explicit arbiter. Bus negotiations must be carried out by software on the microcontroller.

The serial port represents an important asynchronous bit-level device with byte-level controller support. It uses I/O pins that are connected to an internal UART controller. In transmit mode, the UART takes a byte of data and shifts it out serially at a specified interval. In receive mode, it samples the input pin for a transition and shifts in bits at a specified interval from the edge. Interrupts are triggered in the processor to signal completion events.

The coprocessor represents a synchronous bit-level device with byte-level support. In this case, it is a very limited MCU (AT90LS2343 [2], with 2 KB flash instruction memory, 128 bytes of SRAM and EEPROM) that uses I/O pins connected to an SPI controller. SPI is a synchronous serial data link, providing high speed full-duplex connections (up to 1 Mbit) between various peripherals. The coprocessor is connected in a way that allows it to reprogram the main microcontroller. The sensor can be reprogrammed by transferring data from the network into the coprocessor's 256 KB EEPROM (24LC256). Alternatively the main processor can use the coprocessor as a gateway to extra storage.

Future extensions to the design follow two paths: making the design more modular and systematic and adding self-monitoring capabilities. In order to make it more modular, a daughterboard connector will be defined; it will expose several chip-to-chip busses like I^2C and SPI, as well as analog sensor interfaces and power. The self-monitoring capabilities will include sensors for battery strength and radio signal strength, and an actuator for controlling radio transmission strength.

3.2 Power Characteristics

Table 1 shows the current drawn by each hardware component under three scenarios: peak load when active, load in "idle" mode, and inactive. When active, the power consumption of the LED and radio reception are about equal to the processor. The processor, radio, and sensors running at peak load consume 19.5 mA at 3 volts, or about 60 mW. (If all the LEDs are on, this increases to 100 mW.) This figure should be contrasted with the 10 μA current draw in the inactive mode. Clearly, the biggest savings are obtained by making unused components inactive whenever possible. The system must embrace the philosophy of getting the work done as quickly as possible and going to sleep.

Component	Active (mA)	Idle (mA)	Inactive (μA)
MCU core (AT90S8535)	5	2	1
MCU pins	1.5	-	-
LED	4.6 each	-	-
Photocell	.3	-	-
Radio (RFM TR1000)	12 tx	-	5
Radio (RFM TR1000)	4.5 rx	-	5
Temp (AD7416)	1	0.6	1.5
Co-proc (AT90LS2343)	2.4	.5	1
EEPROM (24LC256)	3	-	1

Table 1: Current per hardware component of baseline networked sensor platform. Our prototype is powered by an Energizer CR2450 lithium battery rated at 575 mAh. At peak load, the system consumes 19.5 mA of current, or can run about 30 hours on a single battery. In the idle mode, the system can run for 200 hours. When switched into inactive mode, the system draws only 10 μA of current, and a single battery can run for over a year.

The minimum pulse width for the RFM radio is 52 μs. Thus, it takes on the order of 1 μJ of energy to transmit a single bit[1] and on the order of 0.5 μJ of energy to receive a bit. During this time, the processor can execute 208 cycles (roughly 100 instructions) and can consume up to .8 μJ. A fraction of this instruction count is devoted to bit level processing. The remainder can go to higher level processing (byte-level, packet level, application level) amortized over several bit times. Unused time can be spent in idle or power-down mode.

To broaden the coverage of our study, we deploy these networked sensors in two configurations. One is a mobile sensor that picks up temperature and light readings and periodically presents them on the wireless network as tagged data objects. It needs to conserve its limited energy. The second is a stationary sensor that bridges the radio network through the serial link to a host on the Internet. It has power supplied by its host, but also has more demanding data flows.

4. TINY MICROTHREADING OPERATING SYSTEM (TinyOS)

The core challenge we face is to meet the requirements for networked sensors put forth in Section 2 upon the class of platforms represented by the design in Section 3 in manner that scales forward to future technology. Small physical size, modest active power load and tiny inactive load are provided by the hardware design. An operating system framework is needed that will retain these characteristics by managing the hardware capabilities effectively, while supporting concurrency-intensive operation in a manner that achieves efficient modularity and robustness.

For reasons described in Section 6, existing embedded device operating systems do not meet this challenge. Also, we

[1]Transmitting a one costs 1.9 μJ and transmitting a zero is free. The transmitter requires DC balance (an equal number of ones and zeros), so the precise energy cost per bit is very dependent on the encoding.

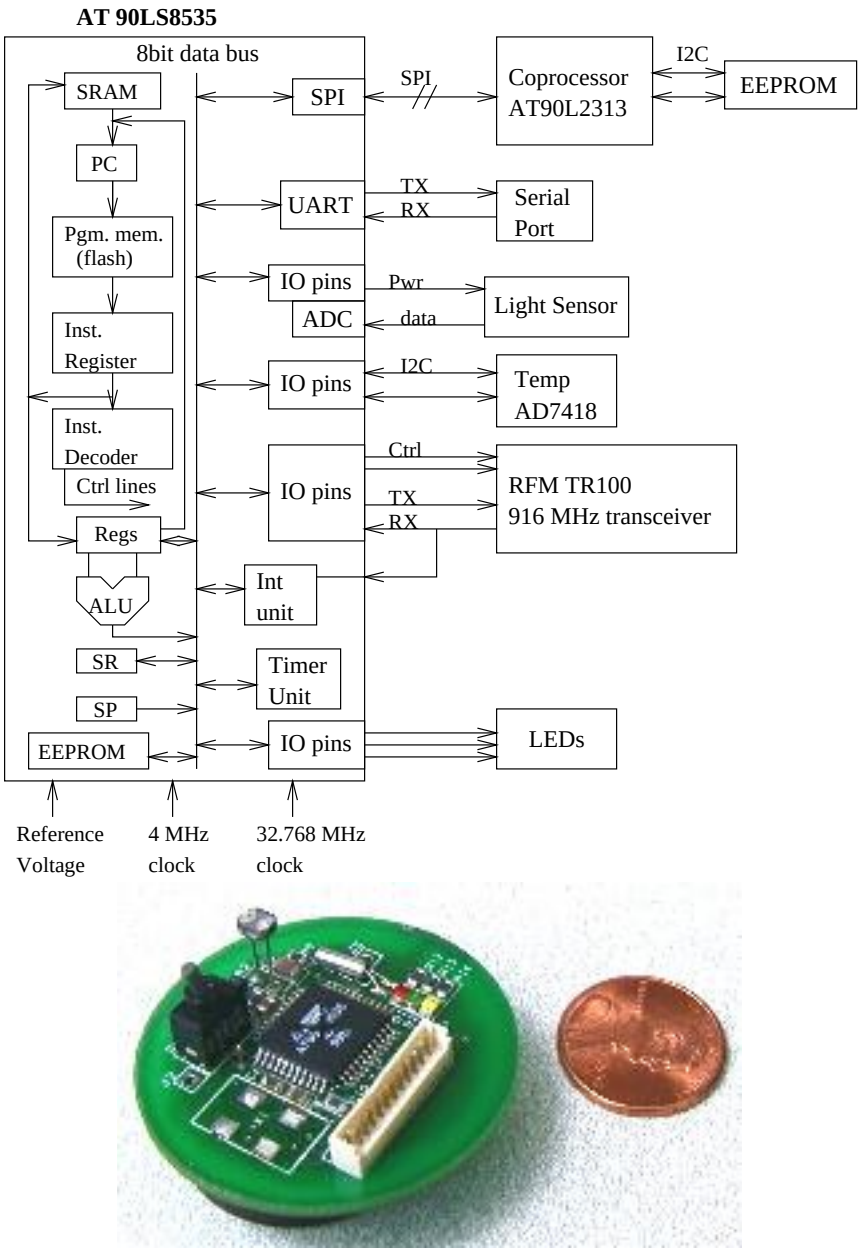


Figure 1: Photograph and schematic for representative network sensor platform

desire a clean open platform to explore alternatives. The problem we must tackle is strikingly similar to that of building efficient network interfaces, which also must maintain a large number of concurrent flows and juggle numerous outstanding events [20]. This has been tackled through physical parallelism [21] and virtual machines [27]. We tackle it by building an extremely efficient multithreading engine. As in TAM [22] and CILK [23] it maintains a two-level scheduling structure, so a small amount of processing associated with hardware events can be performed immediately while long running *tasks* are interrupted. The execution model is similar to FSM models, but considerably more programmable.

Our system is designed to scale with the current technology trends supporting both smaller, tightly integrated designs as well as the crossover of software components into hardware. This is in contrast to traditional notions of scalability that are centered on scaling up total power/resources/work for a given computing paradigm. It is essential that network sensor architectures plan for the eventual integration of sensors, processing and communication. The days of sensor packs being dominated by interconnect and support hardware, as opposed to physical sensors, are numbered.

In TinyOS, we have chosen an event model so that high levels of concurrency can be handled in a very small amount of space. A stack-based threaded approach would require that stack space be reserved for each execution context. Additionally, it would need to be able to multi-task between these execution contexts at a rate of 40,000 switches per second, or twice every 50 μs - once to service the radio and once to perform all other work. It is clear that an event-based regime lends itself to these requirements. It is not surprising that researchers in the area of high performance computing have seen this same phenomena – that event based programming must be used to achieve high performance in concurrency intensive applications [28, 42].

In this design space, power is the most precious resource. We believe that the event-based approach creates a system that uses CPU resources efficiently. The collection of tasks associated with an event are handled rapidly, and no blocking or polling is permitted. Unused CPU cycles are spent in the sleep state as opposed to actively looking for an interesting event. Additionally, with real-time constraints the calculation of CPU utilization becomes simple – allowing for algorithms that adjust processor speed and voltage accordingly [36, 44].

4.1 TinyOS Design

A complete system configuration consists of a tiny scheduler and a graph of *components*. A component has four interrelated parts: a set of *command handlers*, a set of *event handlers*, an encapsulated fixed-size *frame*, and a bundle of simple *tasks*. Tasks, commands, and handlers execute in the context of the frame and operate on its state. To facilitate modularity, each component also declares the commands it uses and the events it signals. These declarations are used to compose the modular components in a per-application configuration. The composition process creates layers of components where higher level components issue commands to lower level components and lower level components signal events to the higher level components. Physical hardware represents the lowest level of components.

The fixed size frames are statically allocated which allows us to know the memory requirements of a component at compile time. Additionally, it prevents the overhead associated with dynamic allocation. This savings manifests itself in many ways, including execution time savings because variable locations can be statically compiled into the program instead of accessing state via pointers.

Commands are non-blocking requests made to lower level components. Typically, a command will deposit request parameters into its frame and conditionally post a task for later execution. It may also invoke lower commands, but it must not wait for long or indeterminate latency actions to take place. A command must provide feedback to its caller by returning status indicating whether it was successful or not, *e.g.*, buffer overrun.

Event handlers are invoked to deal with hardware events, either directly or indirectly. The lowest level components have handlers connected directly to hardware interrupts, which may be external interrupts, timer events, or counter events. An event handler can deposit information into its frame, post tasks, signal higher level events or call lower level commands. A hardware event triggers a fountain of processing that goes upward through events and can bend downward through commands. In order to avoid cycles in the command/event chain, commands cannot signal events. Both commands and events are intended to perform a small, fixed amount of work, which occurs within the context of their component's state.

Tasks perform the primary work. They are atomic with respect to other tasks and run to completion, though they can be preempted by events. Tasks can call lower level commands, signal higher level events, and schedule other tasks within a component. The run-to-completion semantics of tasks make it possible to allocate a single stack that is assigned to the currently executing task. This is essential in memory constrained systems. Tasks allow us to simulate concurrency within each component, since they execute asynchronously with respect to events. However, tasks must never block or spin wait or they will prevent progress in other components. While events and commands approximate instantaneous state transitions, task bundles provide a way to incorporate arbitrary computation into the event driven model.

The task scheduler is currently a simple FIFO scheduler, utilizing a bounded size scheduling data structure. Depending on the requirements of the application, more sophisticated priority-based or deadline-based structures can be used. It is crucial that the scheduler is power aware: our prototype puts the processor to sleep when the task queue is empty, but leaves the peripherals operating, so that any of them can wake up the system. This behavior enables us to provide efficient battery usage (see Section 5). Once the queue is empty, another task can be scheduled only as a result of an event, thus there is no need for the scheduler to wake up until a hardware event triggers activity. More aggressive power management is left to the application.

4.2 Example Component

A typical component including a frame, event handlers, commands and tasks for a message handling component is pictured in Figure 2. Like most components, it exports commands for initialization and power management. Additionally, it has a command for initiating a message transmission, and signals events on the completion of a transmission or the arrival of a message. In order to perform its function,

the messaging component issues commands to a packet level component and handles two types of events: one that indicates a message has been transmitted and one that signals that a message has been received.

Since the components describe both the resources they provide and the resources they require, connecting them together is very simple. The programmer simply matches the signatures of events and commands required by one component with the signatures of events and commands provided by another component. The communication across the components takes the form of a function call, which has low overhead and provides compile time type checking.

4.3 Component Types

In general, components fall into one of three categories: hardware abstractions, synthetic hardware, and high level software components.

Hardware abstraction components map physical hardware into our component model. The **RFM** radio component (shown in lower left corner of Figure 3) is representative of this class. This component exports commands to manipulate the individual I/O pins connected to the RFM transceiver and posts events informing other components about the transmission and reception of bits. Its frame contains information about the current state of the component (the transceiver is in sending or receiving mode, the current bit rate, etc.). The **RFM** consumes the hardware interrupt, which is transformed into either the `RX_bit_evt` or into the `TX_bit_evt`. There are no tasks within the **RFM** because the hardware itself provides the concurrency. This model of abstracting over the hardware resources can scale from very simple resources, like individual I/O pins, to quite complex ones, like UARTs.

Synthetic hardware components simulate the behavior of advanced hardware. A good example of such component is the **Radio Byte** component (see Figure 3). It shifts data into or out of the underlying **RFM** module and signals when an entire byte has completed. The internal tasks perform simple encoding and decoding of the data. [2] Conceptually, this component is an enhanced state machine that could be directly cast into hardware. From the point of view of the higher levels, this component provides an interface and functionality very similar to the UART hardware abstraction component: they provide the same commands and signal the same events, deal with data of the same granularity, and internally perform similar tasks (looking for a start bit or symbol, perform simple encoding, etc.).

The high level software components perform control, routing and all data transformations. A representative of this class is the messaging module presented above, in Figure 2. It performs the function of filling in a packet buffer prior to transmission and dispatches received messages to their appropriate place. Additionally, components that perform calculations on data or data aggregation fall into this category.

Our component model allows for easy migration of the hardware/software boundary. This is possible because our event based model is complementary to the underlying hardware. Additionally, the use of fixed size, preallocated storage is a requirement for hardware based implementations. This ease of migration from software to hardware will be particularly important for networked sensors, where the system designers will want to explore the tradeoffs between the scale of integration, power requirements, and the cost of the system.

[2]The radio requires that the data transmitted is DC-balanced. We currently use Manchester encoding.

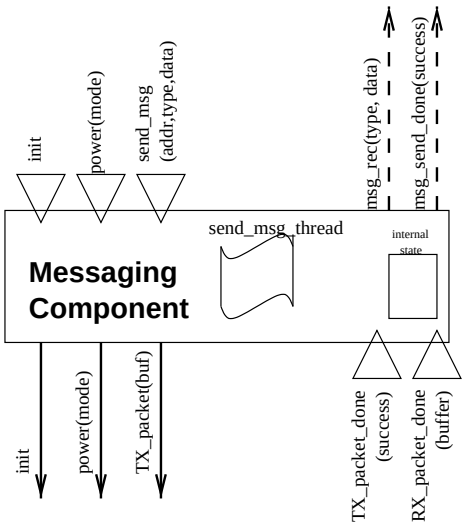
4.4 Putting it all together

Now, that we have shown a few sample components, we will examine their composition and their interaction within a complete configuration. To illustrate the interaction of the components, we describe a networked sensor application we have developed. The application consists of a number of sensors distributed within a localized area. They monitor the temperature and light conditions and periodically transmit their measurements to a central base station. Each sensor not only acts as a data source, but it may also forward data for sensors that are out of range of the base station. In our application, each sensor dynamically determines the correct routing topology for the network. The internal component graph of a base station sensor is shown in Figure 3 along with the routing topology created by a collection of sensors.

There are three I/O devices that this application must service: the network, the light sensor, and the temperature sensor. Each of these devices is represented by a vertical stack of components. The stacks are tied together by the application layer. We chose an abstraction similar to active messages [42] for our top level communication model. The active message model includes handler identifiers with each message. The networking layer invokes the indicated handler when a message arrives. This integrates well with our execution model because the invocation of message handlers takes the form of events being signaled in the application. Our application data is broadcast in the form of fixed length active messages. If the receiver is an intermediate hop on the way to the base station, the message handler initiates the retransmission of the message to the next recipient. Once at the base station, the handler forwards the packet to the attached computer.

The application works by having a base station periodically broadcast out route updates. Any sensors in range of this broadcast record the identity of the base station and then rebroadcast out the update. Each sensor remembers the first update that is received in an era, and uses the source of the update as the destination for routing data to the base station. Each device also periodically reads its sensor data and transmits the collected data towards the base station. At the high level, there are three significant events that each device must respond to: the arrival of a rout update, the arrival of a message that needs to be forwarded, and the collection of new data.

Internally, when our application is running, thousands of events are flowing through each sensor. A timer event is used to periodically start the data collection. Once the temperature and light information have been collected, the application uses the messaging layer's `send_message` command to initiate a transfer. This command records the message location in the **AM** component's frame and schedules a task to handle the transmission. When executed, this task composes a packet, and initiates a downward chain of commands by calling the `TX_packet` command in the **Packet** component. In turn, the command calls `TX_byte` within the **Radio Byte** component to start the byte-by-byte transmission. The **Packet** component internally acts as a data drain, handing bytes down to the **Radio Byte** component




```
/* Messaging Component Declaration */

//ACCEPTS:
char TOS_COMMAND(AM_send_msg)(int addr,int type,
                              char* data);
void TOS_COMMAND(AM_power)(char mode);
char TOS_COMMAND(AM_init)();
//SIGNALS:
char AM_msg_rec(int type, char* data);
char AM_msg_send_done(char success);
//HANDLES:
char AM_TX_packet_done(char success);
char AM_RX_packet_done(char* packet);
//USES:
char TOS_COMMAND(AM_SUB_TX_packet)(char* data);
void TOS_COMMAND(AM_SUB_power)(char mode);
char TOS_COMMAND(AM_SUB_init)();
```

Figure 2: A sample messaging component. Pictorially, we represent the component as a bundle of tasks, a block of state (component frame) a set of commands (upside-down triangles), a set of handlers (triangles), solid downward arcs for commands they use, and dashed upward arcs for events they signal. All of these elements are explicit in the component code.

whenever the previous byte transmission is complete. Internally, **Radio Byte** prepares for transmission by putting the **RFM** component into the transmission state (if appropriate) and scheduling the `encode_task` to prepare the byte for transmission. When the `encode_task` is scheduled, it encodes the data, and sends the first bit of data to the **RFM** component for transmission. The **Radio Byte** also acts as a data drain, providing bits to the **RFM** in response to the `TX_bit_evt` event. If the byte transmission is complete, then the **Radio Byte** will propagate the `TX_bit_evt` signal to the packet-level controller through the `TX_byte_done` event. When all the bytes of the packet have been drained, the packet level will signal the `TX_packet_done` event, which will signal the the application through the `msg_send_done` event.

When a transmission is not in progress, and the sensor is active, the **Radio Byte** component receives bits from the **RFM** component. If the start sequence is detected, the transmission process is reversed: bits are collected into bytes and bytes are collected into packets. Each component acts as a data-pump: it actively signals the incoming data to the higher levels of the system, rather than respond to a read operation from above. Once a packet is available, the address of the packet is checked and if it matches the local address, the appropriate handler is invoked.

Component Name	Code Size (bytes)	Data Size (bytes)
Multihop router	88	0
AM_dispatch	40	0
AM_temperature	78	32
AM_light	146	8
AM	356	40
Packet	334	40
RADIO_byte	810	8
RFM	310	1
Photo	84	1
Temperature	64	1
UART	196	1
UART_packet	314	40
I2C_bus	198	8
Procesor_init	172	30
TinyOS scheduler	178	16
C runtime	82	0
Total	3450	226

Table 2: Code and data size breakdown for our complete system. Only the processor init, the TinyOS scheduler, and the C runtime are required for every application, the other components are included as needed.

5. EVALUATION

Small physical size: Table 2 shows the code and data size for each of the components in our system. It is clear that the code size of our complete system, including a network sensor application with simple multi-hop routing, is remarkable. In particular, our scheduler only occupies 178 bytes and our complete network sensor application requires only about 3KB of instruction memory. Furthermore, the data size of our scheduler is only 16 bytes, which utilizes only 3% of the available data memory. Our entire application comes in at 226 bytes of data, still under 50% of the 512 bytes available.

Concurrency-intensive operations: As we argued in Section 2, network sensors need to handle multiple flows of information simultaneously. In this context, an important baseline characteristic of a network sensor is its context switch speed. Table 3 shows this aspect calibrated against the intrinsic hardware cost for moving bytes in memory. The cost of propagating an event is roughly equivalent to that of copying one byte of data. This low overhead is essential for achieving modular efficiency. Posting a task and switching context costs about as much as moving 6 bytes of memory.

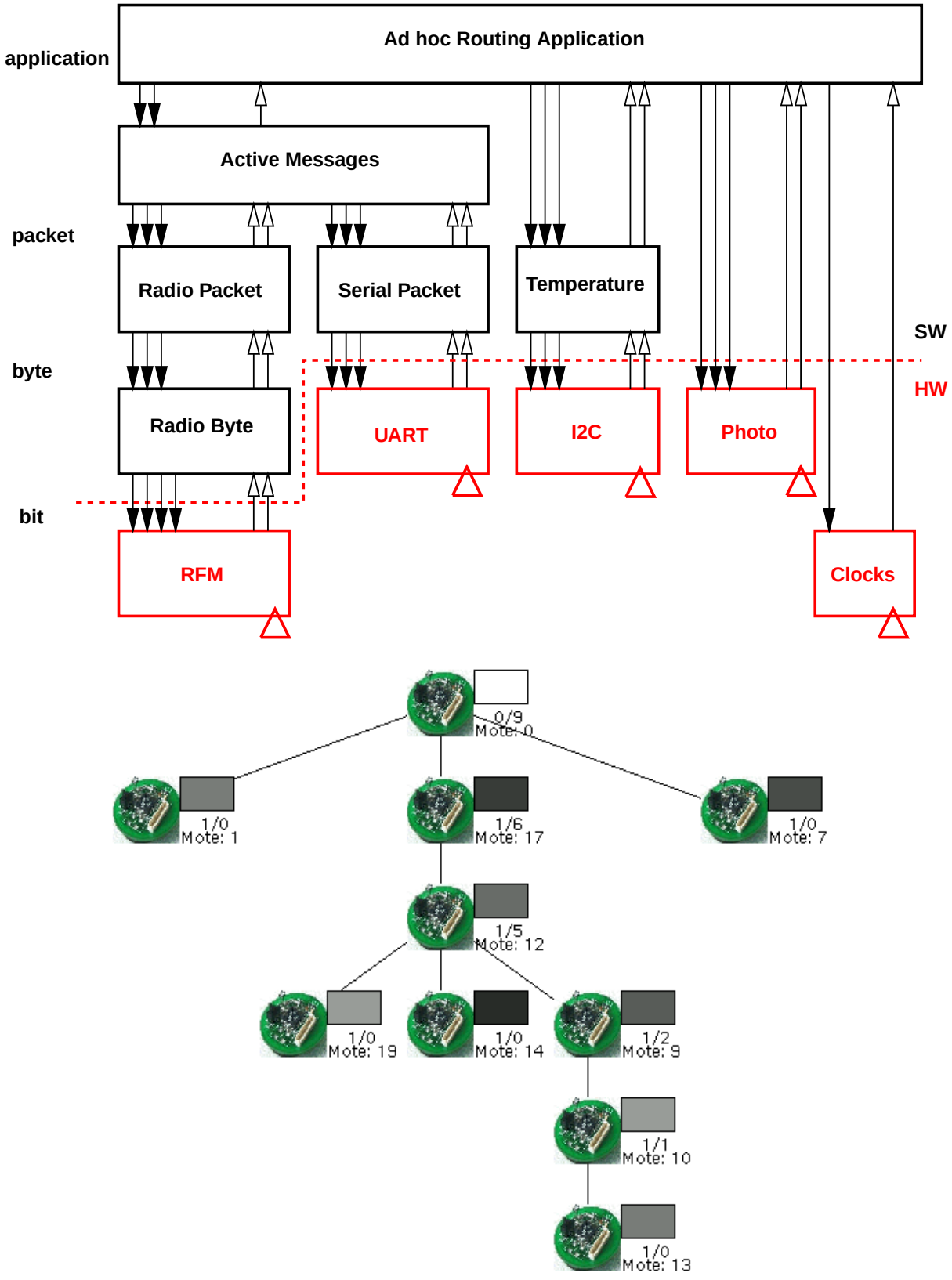


Figure 3: A sample configuration of a networked sensor, and the routing topology created by a collection of distributed sensors.

Operations	Cost (cycles)	Time (μs)	Normalized to byte copy
Byte copy	8	2	1
Post an Event	10	2.5	1.25
Call a Command	10	2.5	1.25
Post a task to scheduler	46	11.5	6
Context switch overhead	51	12.75	6
Interrupt (hardware cost)	9	2.25	1
Interrupt (software cost)	71	17.75	9

Table 3: Overhead of primitive operations in TinyOS

Our most expensive operation involves the low-level aspects of interrupt handling. Though the hardware operations for handling interrupts are fast, the software operations that save and restore registers in memory impose a significant overhead. Several techniques can be used to reduce that overhead: partitioning the register set [22] or use of register windows [14].

Efficient modularity: One of the key characteristics of our systems is that events and commands can propagate through components quickly. Projects such as paths, in Scout [35], and stackable systems [29, 25, 24] have had similar goals in other regimes. Table 3 gives the cost of individual component crossing, while Figure 4 shows the dynamic composition of these crossings. It contains a timing diagram from a logic analyzer of an event chain that flows through the system at the completion of a radio transmission. The events fire up through our component stack eventually causing a command to transmit a second message. The total propagation delay up the five layer radio communication stack is 40 μs or about 80 instructions. This is discussed in detail in Figure 4; steps 0 through 4 show the event crossing these layers. The entire event propagation delay plus the cost of posting a command to schedule a task to send the next packet (step 0 through 6) is about 90 μs.

Limited physical parallelism and controller hierarchy: We have successfully demonstrated a system managing multiple flows of data through a single microcontroller. Table 4 shows the work and energy distribution among each of our software components while engaged in active data transmission. Even during this highly active period, the processor is idle approximately 50% of the time. The remaining time can be used to access other sensors, like the photo sensor, or the I^2C temperature controller. Even if other I/O devices provide an interface as primitive as our radio, a single controller can support flows of data at rates up to 40 μs per bit or 25Kbps. Furthermore, this data can be used to make design choices about the amount of physical parallelism necessary. For example, while the low level bit and byte processing utilize significant CPU resources, the CPU is not the system bottleneck. If bit level functions were implemented on a separate microcontroller, we would not realize a performance gain because of the radio bandwidth limitations. We would also incur additional power and time expense in transferring data between microcontrollers. However, if these components were implemented by dedicated hardware, we would be able to make several power saving design choices including sleeping, which would save 690 μJ per bit, or lowering the frequency of the processor 20-fold.

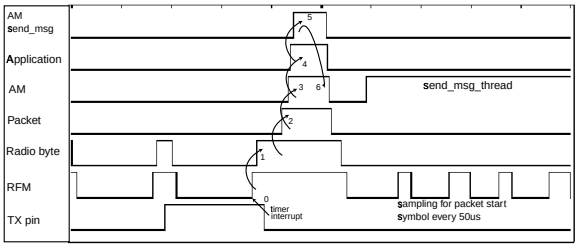


Figure 4: A timing diagram from a logic analyzer capturing event propagation across networking components at a granularity of 50 μs per division. The graph shows the send message scenario described in Section 4.4 focusing on transmission of the last bit of the packet. Starting from the hardware timer interrupt of step 0, events propagate up through the `TX_bit_evt` in step 1, into byte-level processing. The handler issues a command to transmit the final bit and then fires the `TX_byte_ready` event in step 2 to signal the end of the byte. This triggers `TX_packet_done` in step 3. Step 4 signals the application that the `send_msg` command has finished. The application then issues another asynchronous `send_msg` command in step 5 which post a task at step 6 to send the packet. While `send_msg_task` prepares the message, the RFM component is periodically scheduled to listen for incoming packets. The event propagation delay from step 0 to step 4 is about 40 μs while for the entire event and command fountain starting from step 0 to step 6 to be completed, the total elapsed time is about 95 μs.

Diversity in usage and robust operation: Finally, we have been able to test the versatility of this architecture by creating sample applications that exploit the modular structure of our system. These include source based multi-hop routing applications, active-badge-like [43] location detection applications and sensor network monitoring applications. Additionally by developing our system in C, we have the ability to target multiple CPU architectures in future systems. Furthermore, our multi-hop routing application automatically reconfigures itself to withstand individual node failure so that the sensor network as a whole is robust.

6. RELATED WORK

There is a large amount of work on developing micro-electromechanical sensors and new communication devices [38, 37]. The development of these new devices make a strong case for the development of a software platform to support and connect them. TinyOS is designed to fill this role. We believe that current real-time operating systems do not meet the needs of this emerging integrated regime. Many of them have followed the performance growth of the wallet size device.

Traditional real time embedded operating systems include VxWorks [13], WinCE [19], PalmOS [4], and QNX [26] and many others [8, 32, 34]. Table 5 shows the characteristics for a handful of these systems. Many are based on microkernels that allow for capabilities to be added or removed based on system needs. They provide an execution environment that is similar to traditional desktop systems.

Name	Preemption	Protection	ROM Size	Configurable	Targets
pOSEK	Tasks	No	2K	Static	Microcontrollers
pSOSystem	POSIX	Optional		Dynamic	PII → ARM Thumb
VxWorks	POSIX	Yes	≈ 286K	Dynamic	Pentium → Strong ARM
QNX Neutrino	POSIX	Yes	$> 100K$	Dynamic	Pentium II → NEC chips
QNX Realtime	POSIX	Yes	100K	Dynamic	Pentium II → 386's
OS-9	Process	Yes		Dynamic	Pentium → SH4
Chorus OS	POSIX	Optional	10K	Dynamic	Pentium → Strong ARM
Ariel	Tasks	No	19K	Static	SH2, ARM Thumb
CREEM	data-flow	No	560 bytes	Static	ATMEL 8051

Table 5: A comparison of selected architecture features of several embedded OSes.

Components	Packet reception breakdown	Percent CPU Utilization	Energy (nJ/bit)
AM	0.05%	0.02%	0.33
Packet	1.12%	0.51%	7.58
Radio handler	26.87%	12.16%	182.38
Radio decode task	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66
Components	Packet transmission breakdown	Percent CPU Utilization	Energy (nJ/bit)
AM	0.03%	0.01%	0.18
Packet	3.33%	1.59%	23.89
Radio handler	35.32%	16.90%	253.55
Radio encode task	4.53%	2.17%	32.52
RFM	56.80%	27.18%	407.17
Radio Transmission	-	-	1800
Idle	-	52.14%	-
Total	100.00%	100.00%	4317.89

Table 4: Details breakdown of work distribution and energy consumption across each layer for packet transmission and reception. For example, 66.48% of the work in receiving packets is done in the RFM bit-level component and it utilizes 30.08% of the CPU time during the entire period of receiving the packet. It also consumes 451.17nJ per bit it processes. Note that these measurements are done with respect to raw bits at the physical layer with the bit rate of the radio set to 100 μs/bit using DC-balanced ON-OFF keying.

Their POSIX [40] compatible thread packages allow system programmers to reuse existing code and multiprogramming techniques. The largest RTOSs provide memory protection given the appropriate hardware support. This becomes increasingly important as the size of the embedded applications grow. In addition to providing fault isolation, memory protection prevents corrupt pointers from causing seemingly unrelated errors in other parts of the program allowing for easier software development. These systems are a popular choice for PDAs, cell phones and set-top-boxes. However, they do not come close to meeting our requirements; they are more suited to the world of embedded PCs. For example, a QNX context switch requires over 2400 cycles on a 33MHz 386EX processor, and the memory footprint of VxWorks is in the hundreds of kilobytes. [3] Both of these statistics are more than an order of magnitude beyond our required limits.

There is also a collection of smaller *real time executives* including Creem [30], pOSEK [7], and Ariel [3], which are minimal operating systems designed for deeply embedded systems, such as motor controllers or microwave ovens. While providing support for preemptive tasks, they have severely constrained execution and storage models. pOSEK, for example, provides a task-based execution model that is statically configured to meet the requirements of a specific application. Generally, these systems approach the space requirements and represent designs closest to ours. However, they tend to be control centric – controlling access to hardware resources – as opposed to dataflow-centric. Even the pOSEK, which meets our memory requirements, exceeds the limitations we have on context switch time. At its optimal performance level and with the assumption that the CPI and instructions per program of the PowerPC are equivalent to that of the 8-bit ATMEL the context switch time would be over 40 μs.

Other related work includes [17] where a finite state machine (FSM) description language is used to express component designs that are compiled down to software. However, they assume that this software will then operate on top of a real-time OS that will give them the necessary concurrency. This work is complementary to our own in that the requirements of an FSM based design maps well onto our event/command structure. We also have the ability to support the high levels of concurrency inherent in many finite state machines.

On the device side, [6] is developing a cubic millimeter integrated network sensors. Additionally, [38, 15] has developed low power hardware to support the streaming of sensor readings over wireless communication channels. In their work, they explicitly mention the need for the inclusion of a microcontroller and the support of multi-hop routing. Both of these systems require the support of an efficient software architecture that allows high levels of concurrency to manage communication and data collection. Our system is designed to scale down to the types of devices they envision.

A final class of related work is that of applications that will be enabled by networked sensors. Piconet [16] and The Active Badge Location System [43] have explored the utility of networked sensors. Their applications include per-

[3]It is troubling to note that while there is a large amount of information on code size of embedded OSes, there are very few hard performance numbers published. [9] has started a program to test various real-time operating systems yet they are keeping the results confidential - you can view them for a fee.

sonnel tracking and information distribution from wireless, portable communication devices. However, they have focused on the applications of such devices as opposed to the system architecture that will allow a heterogeneous group of devices to scale down to the cubic millimeter category.

7. ARCHITECTURAL IMPLICATIONS

A major architectural question in the design of network sensors is whether or not individual microcontrollers should be used to manage each I/O device. We have demonstrated that it is possible to maintain multiple flows of data with a single microcontroller. This shows that it is an architectural option - not a requirement - to utilize individual microcontrollers per device. Moreover, the interconnect of such a system will need to support an efficient event based communication model. Tradeoffs quickly arise between power consumption, speed of off chip communication, flexibility and functionality. Additionally, our quantitative analysis has enabled us to consider the effects of using alternative microcontrollers. We believe that the use of a higher performance ARM Thumb [1] would not change our architecture. Furthermore, our architecture allows us to calculate the minimum performance requirements of a processor. Along similar lines, we can extrapolate how our technology will perform in the presence of higher speed radio components. It is clear that bit level processing cannot be used with the transfer rates of Bluetooth radios [11]; the **Radio Byte** component needs to become a hardware abstraction rather than synthetic hardware.

Further analysis of our timing breakdown in Table 4 can reveal the impact of architectural changes in microcontrollers. For example, the inclusion of hardware support for events would make a significant performance impact. An additional register set for the execution of events would save us about 20 μs per event or about 20% of our total CPU load. This savings could be directly transferred to either higher performance or lower power consumption.

Additionally, we are able to quantify the effects of additional hardware support for managing data transmission. Table 4 shows that hardware support for the byte level collection of data from the radio would save us a total of about 690 μJ per bit in processor overhead. This represents the elimination of the bit level processing from the CPU. Extension of this analysis can reveal the implication of several other architectural changes including the use of radios that can automatically wake themselves at the start of an incoming transmission or a hardware implementation of a MAC layer.

Furthermore, the impact of reconfigurable computing can be investigated relative to our design point. In traditional systems, the interconnect and controller hierarchy is configured for a particular system niche, where as in future network sensors it will be integrated on chip. Reconfigurable computing has the potential of making integrated network sensors highly versatile. The **Radio Byte** component is a perfect candidate for reconfigurable support. It consumes a significant amount of CPU time and must be radio protocol specific. A standard UART or DMA controller is much less effective in this situation because the component must search for the complex start symbol prior to clocking in the bits of the transmission. However, it could be trivially implemented in a FPGA.

All of this extrapolation was made possible by fully developing and analyzing quantitatively a specific design point for a network sensor. It is clear that there is a strong tie between the software execution model and the hardware architecture that supports it. Just as SPEC benchmarks attempted to evaluate the impact of architectural changes on the entire system in the workstation regime, we have attempted to begin the systematic analysis architectural alternatives in the network sensor regime.

8. REFERENCES

[1] Atmel AT91 Arm Thumb. `http://www.atmel.com/atmel/products/prod35.htm`.

[2] Atmel AVR 8-Bit RISC processor. `http://www.atmel.com/atmel/products/prod23.htm`.

[3] Microware Ariel Technical Overview. `http://www.microware.com/ProductsServices/Technologies/ariel_technology_brief.html`.

[4] PalmOS Software 3.5 Overview. `http://www.palm.com/devzone/docs/palmos35.html`.

[5] Pico Radio. `http://bwrc.eecs.berkeley.edu/Research/Pico_Radio/`.

[6] Pister, K.S.J. Smart Dust. `http://www.atmel.com/atmel/products/prod23.htm`.

[7] pOSEK, A super-small, scalable real-time operating system for high-volume, deeply embedded applications. `http://www.isi.com/products/posek/index.htm`.

[8] pSOSystem Datasheet. `http://www.windriver.com/products/html/psosystem_ds.html`.

[9] Real-Time Consult. `http://www.realtime-info.com/encyc/market/rtos/eval_introduction.htm`.

[10] RF Monolithics. `http://www.rfm.com/products/data/tr1000.pdf`.

[11] The Official Bluetooth Website. `http://www.bluetooth.com`.

[12] uClinux, The Linux/Microcontroller Project. `http://www.uclinux.org/`.

[13] VxWorks 5.4 Datasheet. `http://www.windriver.com/products/html/vxwks54_ds.html`.

[14] Anant Agarwal, Geoffrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Daniel Nussbaum, Mike Parkin, and Donald Yeung. The MIT alewife machine : A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic, 1991.

[15] B. Atwood, B.Warneke, and K.S.J. Pister. Preliminary circuits for smart dust. In *Proceedings of the 2000 Southwest Symposium on Mixed-Signal Design*, San Diego, California, February 27-29 2000.

[16] F. Bennett, D. Clarke, J. Evans, A. Hopper, A. Jones, and D. Leask. Piconet: Embedded mobile networking, 1997.

[17] M. Chiodo. Synthesis of software programs for embedded control applications, 1995.

[18] Chu, P.B., Lo, N.R., Berg, E., Pister, K.S.J. Optical communication link using micromachined corner cuber reflectors. In *Proceedings of SPIE vol.3008-20.*, 1997.

[19] Microsoft Corp. Microsoft Windows CE. `http://www.microsoft.com/windowsce/embedded/`.

[20] D. Culler, J. Singh, and A. Gupta. Parallel computer

architecture a hardware/software approach, 1999.

[21] R. Esser and R. Knecht. Intel Paragon XP/S – architecture and software environment. Technical Report KFA-ZAM-IB-9305, 1993.

[22] D. Culler et. al. Fine grain parallelism with minimal hardware support: A compiler-controlled treaded abstract machine. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[23] R.D. Blumofe et. al. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[24] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the ficus replicated file system. In *Proceedings of the Summer USENIX Conference*, pages pages 63–71, Anaheim, CA, June 1990.

[25] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. In *ACM Transactions on Computer Systems*, pages 12(1):58–89, Feb. 1994.

[26] Dan Hildebrand. An Architectural Overview of QNX. `http://www.qnx.com/literature/whitepapers/archoverview.html`.

[27] M. Homewood and M. McLaren. Meiko cs-2 interconnect elan-elite design, 1993.

[28] James Hu, Irfan Pyarali, and Douglas C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *In Proceedings of the 2 nd Global Internet Conference. IEEE*, November 1997.

[29] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. In *IEEE Transactions on Software Engineering*, pages 17(1):64–76, Jan. 1991.

[30] Barry Kauler. CREEM Concurrent Realitme Embedded Executive for Microcontrollers. `http://www.goofee.com/creem.htm`.

[31] J. Kymissis, C. Kendall, J. Paradiso, and N. Gershenfeld. Parasitic power harvesting in shoes. In *Proc. of the Second IEEE International Conference on Wearable Computing (ISWC), IEEE Computer Society Press*, pages pp. 132–139, October 1998.

[32] QNX Software Systems Ltd. QNX Neutrino Realtime OS . `http://www.qnx.com/products/os/neutrino.html`.

[33] James McLurkin. Algorithms for distributed sensor networks. In *Masters Thesis for Electrical Engineering at the Univeristy of California, Berkeley*, December 1999.

[34] Microware. Microware OS-9. `http://www.microware.com/ProductsServices/Technologies/os-91.html`.

[35] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. Scout: A communications-oriented operating system. In *Hot OS*, May 1995.

[36] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. Int'l Symposium on Low Power Electronics and Design*, pages pp. 76–81, Aug. 1998.

[37] K. S. J. Pister, J. M. Kahn, and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes, 1999.

[38] G. Pottie, W. Kaiser, L. Clare, and H. Marcy. Wireless integrated network sensors, 1998.

[39] Philips Semiconductors. The i^2c-bus specification, version 2.1. `http://www-us.semiconductors.com/acrobat/various/I2C_BUS_SPECIFICATION_3.pdf`, 2000.

[40] I. Standard. Real-time extensions to posix, 1991.

[41] EMJ EMBEDDED SYSTEMS. White Dwarf Linux. `http://www.emjembedded.com/linux/dimmpc.html`.

[42] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation, 1992.

[43] R. Want and A. Hopper. Active badges and personal interactive computing objects, 1992.

[44] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23.