

A wireless embedded sensor architecture for system-level optimization

Jason Hill and David Culler
{jhill, culler}@cs.berkeley.edu

Abstract

Emerging low power, embedded, wireless sensor devices are targeting a wide range of applications, yet have very limited processing, storage, and energy resources. An architecture must be developed that can efficiently meet system demands while simultaneously remaining flexible to application specific optimizations. Analysis of past designs identifies core architectural issues and their impact on system performance. This paper outlines these issues and presents a generalized architecture designed to provide efficient communication mechanisms while allowing for system-level optimizations. The importance of providing a tight coupling between application and communication processing is shown and the tradeoffs associated with virtual versus physical parallelism are investigated. We conclude that a shared controller closely integrated with special-purpose hardware accelerators is the preferred building block for a flexible yet efficient node. A subset of the architecture is implemented using commercial building blocks and shows substantial improvements in communication performance and the ability to perform system-level optimizations that obtain tight synchronization and low power channel monitoring.

1 Introduction

Emerging wireless embedded sensors combine sensing, computation, and communication in a single, tiny, resource constrained device. Their power lies in the ability to deploy a large number of nodes into a physical environment that automatically configure into a distributed sensing platform. Usage scenarios for these devices range from real-time tracking, to monitoring of environmental conditions, to ubiquitous computing environments, to *in situ* monitoring of the health of structures or equipment. While often referred to as networked sensors, they can also be actuators that extend the reach of cyberspace into the physical world.

The core design challenge for wireless embedded sensors lies in coping with their harsh resource constraints. Embedded processors with kilobytes of memory are used to implement complex, distributed, ad-hoc networking protocols. These constraints derive from the vision that these devices will be produced in vast

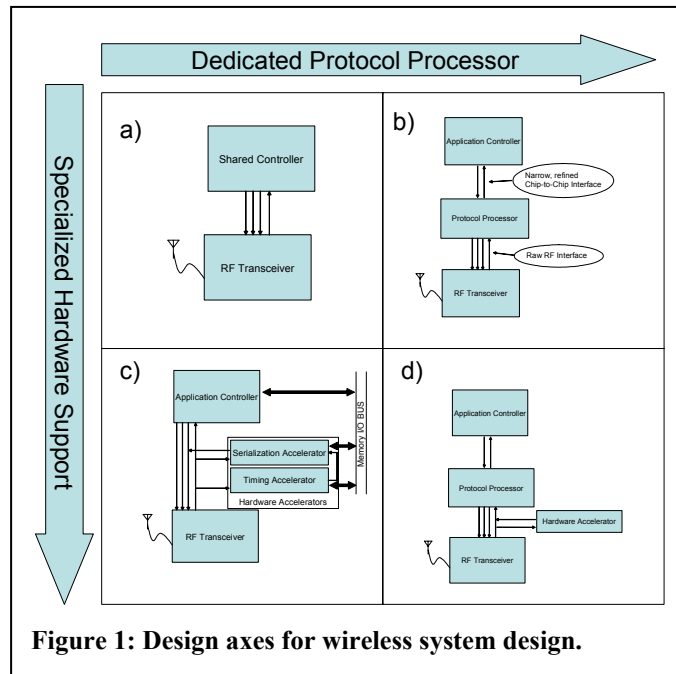
quantities, will be small and inexpensive, and ideally will operate off of ambient power. As Moore's law marches on, these devices will get smaller, not just grow more powerful at a given size.

One of the most difficult resource constraints to meet in the context of wireless embedded sensors is power consumption. As physical size decreases, so does energy capacity. Underlying energy constraints end up creating computational and storage limitations. In these highly constrained devices, a new set of architectural issues has arisen. Many devices, such as cell phones and pagers, use specialized communication hardware in ASICs that provide ultra-low-power implementations of necessary protocol processing [1]. This is possible because they target a narrow set of well-defined applications. The complex tradeoffs between power consumption, bandwidth, and latency can be explored and evaluated with respect to specific performance goals and the optimal protocol can be implemented in hardware. However, the strength of networked embedded devices is their flexibility and universality. The wide range of applications being targeted makes it difficult to develop general-purpose protocols that are efficient for all applications.

This paper outlines a set of recurring issues the architecture of a wireless embedded sensor nodes. While a significant amount of work has focused on the design of low-power radio circuits, reconfigurable logic, and data path optimizations, our focus is on a general architecture for combining these components into a system-level architecture. For efficiency, we believe it is essential to provide a tight coupling between protocol and application level processing. However, traditional processors are inefficient at performing support for key low-level communication operations. While the use of software is essential in creating flexible protocols, we also identify the need to include a hardware support for primitive communication operations that cannot be implemented efficiently in software. Included as hardware accelerators, this minimal hardware support enables the construction of a system that is both efficient and flexible.

Section 2 presents an overview of the core architectural issues in wireless embedded system design. Section 3 presents our generalized architecture that addresses these issues. Section 4 presents the instantiation of our general architecture into a physical device and Section 5 analyzes the implementation. Section 6 overviews the related work and Section 7 concludes with future directions.

The main contributions of this work are (i) a detailed analysis of core issues in low-power wireless communication, (ii) a generalized architecture that addresses these issues, (iii) an instantiation of the architecture using current microcontroller and low-power radio technology, and (iv) an analysis of the instantiated platform.



2 Wireless design issues

In the process of working with several generations of embedded wireless devices, three recurring issues have emerged: 1) How should the concurrency demands of protocol and application-level processing be handled. 2) How to exploit both low-power processing techniques and low-power communication mechanisms. 3) How to define the nature of the interface between applications and their communication protocols.

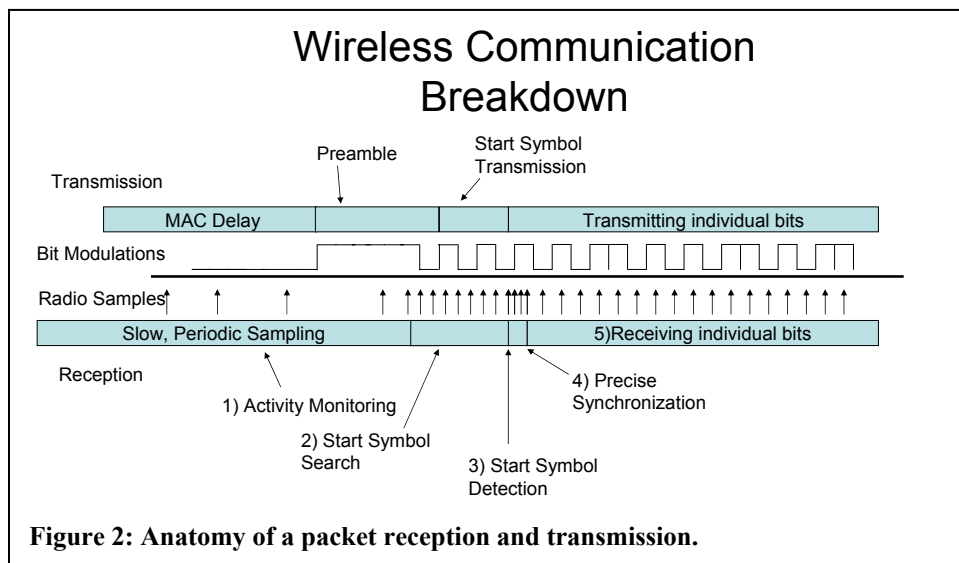
Figure 1 illustrates the basic axes of the design space for wireless devices. The vast majority of wireless devices include a dedicated radio controller that handles all protocol processing. 802.11 cards, Bluetooth chipsets [2], the BWRC PicoRadio [3] and cell phones [1] all use this approach. A protocol processor handles the real-time requirements of modulating and demodulating the radio channel. It refines the raw data stream coming from the radio into a formatted packet interface. For example, the host channel interface (HCI) of Bluetooth deals with high-level packet operations over a UART interface. The intricacies of packet synchronization, channel encoding and media access control (MAC) protocols are all hidden from the application.

The inclusion of a dedicated processor does not reduce the amount of computation performed; it simply confines it to a secondary processor. It also provides a concurrency mechanism so that application and protocol processing can occur simultaneously. The packet processor is a system resource permanently dedicated to radio

processing. However, this strict partitioning of resources leads to non-optimal resource utilization. Additionally, inefficient chip-to-chip communication mechanisms available on current microcontrollers are used to link the protocol processor to applications.

An alternative to the protocol processor based approach is represented by the “motes” in [4]. Instead of using a dedicated packet processor, it time-shares a single execution engine across application and protocol processing. The concurrency requirements of the system are met virtually instead of physically. This allows the computational resources of the system to be dynamically partitioned between application and protocol processing and avoids the overhead of shuttling packets between the processors, as well as the need to define a fixed interface. We show that the benefit of dynamic resource allocation can outweigh the overhead of using virtual parallelism. Additionally, in a virtually partitioned system, the interface between protocol and application processing is purely in software. This allows for rich interfaces to be constructed that can give applications tight control over the communication protocols. A rich interface between applications and their protocols can enable a host of system level optimizations. However, specialized hardware can perform certain low-level radio support far more efficiently than microcontroller processing.

To explore these design issues, we begin by analyzing the performance of two hypothetical systems on the protocol processor axis: a single central controller that is time sliced between application and protocol processing and a dedicated packet controller built out of similar components. To provide a framework for this analysis, a quick outline of the operations that are fundamental to wireless communication is illustrated in Figure 2. 1) Packet



reception starts with monitoring the RF channel for activity. To minimize energy usage periodic sampling is used to detect channel activity that may be a preamble to a start symbol. 2) The start symbol is a unique sequence that marks the beginning of the packet and provides timing information for the packet. For efficiency, the start symbol detection may occur in phases so that general channel activity is detected first at a low cost and then the higher sampling rates are used to precisely detect the start symbol as it arrives. 3) Upon detection of the start symbol, the receiver must attempt to (4) determine the exact timing of the packet so that it can (5) sample the data payload at the center of each bit until the end of the packet. The higher the data rate, the more precise the required packet timing, because the bit widths are smaller. Data samples must be taken at the middle of each bit window throughout the reception.

On the transmission side, the sender must first wait for the channel to become idle by using a media access control (MAC) protocol. It then claims the channel and begins transmitting the preamble and start symbol, followed by the data. Each bit of the transmission must be precisely timed so that the receiver can remain synchronized with the transmitter. The data that is transmitted over the channel is actually an encoded version of the application data. The encoding mechanisms used meet the operating requirements of the radio, e.g. DC balance, and attempt to reduce error probability and to correct bit errors when they occur (forward error correction).

2.1 Concurrency Demands

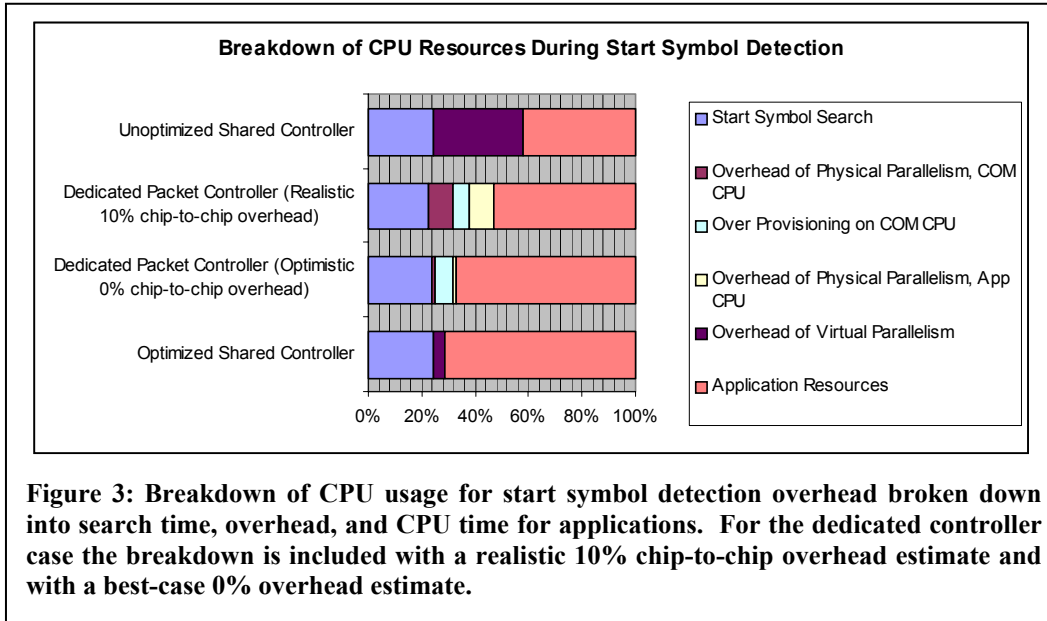
To expose the performance issues associated with the concurrency mechanisms in our two design points, we will analyze the performance of the most demanding aspect of figure 2, start symbol detection. Because start symbol detection must be performed in parallel with application-level processing, it stresses the concurrency mechanisms. It is also a CPU intensive operation that has real-time requirements. For analysis we break the CPU work down into three categories: search time, which is the cycles dedicated to performing the start symbol search, application time, which is cycles that the application can decide how to use, and overhead. In the case of the shared CPU system overhead is the cost of performing context switches. In the dedicated protocol processor system overhead is the cost of the chip-to-chip communication and the over-provisioning required to meet worst-case deadlines.

To compare these two design points we fix the start symbol transmission rate at 10 Kbps and analyze the CPU usage of both systems assuming the same microcontroller is used. For the shared CPU system, we use a single

4 MHz microcontroller. In the dedicated protocol processor based system, we size the protocol processor to just meet the worst-case timing requirements of the start symbol search and then size the application processor so that the system has the equivalent processing of the 4 MHz system.

The start symbol search used in TinyOS [4] requires 49 cycles per sample on average, but has a worst-case overhead of 62 cycles. To detect a start symbol arriving at 10Kbps, a sample must be taken every 50 us. To perform this operation on a dedicated protocol processor, it must be sized to meet the worst-case requirements of the start symbol search. Assuming optimistically that the cost of the chip-to-chip interface is zero, the protocol processor must be able to perform the 62 cycle worst case sample every 50 us, so it operates at 1.25 MHz. Therefore, the application processor may run at 2.75 MHz. In actuality, we expect that the communication between the host CPU and the packet controller would consume a significant amount of resources, which means that more of the CPU resources would need to be allocated to the dedicated to the protocol processor and a portion of the slower application processor would be dedicated to overhead.

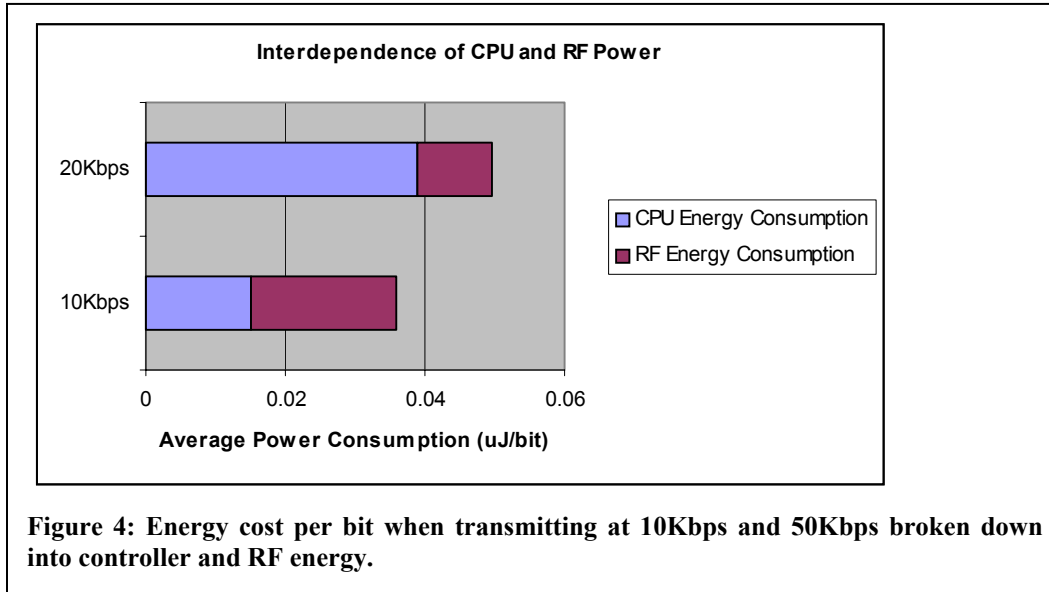
On the shared CPU system, the context switch overhead must be included for each sample. Without any optimizations, 116 cycles are used per sample on average. This leaves just 42% of the 4 MHz processor (1.68 MHz) available for the application as shown in the top of Figure 3. 33.5% of the CPU resources are spent on context switch overhead. In contrast, 6.5% of the CPU resources of the dedicated processor system are spent on the over-provisioning overhead. However, while the over-provisioning is a fundamental part of the partitioned system, context switch overhead is something we can optimize.



We can exploit dynamic resource allocation to reduce the average context switch overhead in the shared case. So far, our analysis is based on having both implementations process every sample in real time. While it is essential that a design be able to process the important bits in real-time, not all bits have to be processed in that fashion. The first half of the start symbol can be detected lazily and then switch to reading each sample in real-time. If just 8 samples could be processed in batch, the average context switch overhead would be reduced by a factor of 8. This would result in a 4.19% context switch overhead, as shown at the bottom of Figure 3, which is less than the 6.5% over provisioning overhead required in the case of the dedicated controller. Because processing the samples in batch does not change the worst-case requirements, the dedicated controller does not benefit from this optimization.

This analysis suggests that the widespread use of dedicated protocol processors is difficult to justify on performance or utilization grounds, especially when the natural dynamic partitioning of resources in the shared case is used to improve overall system performance. While the fixed overhead of context switching can be significant, techniques can be used to properly amortize it. The intermittent nature of the real-time requirements of wireless protocols makes it beneficial to have dynamic resources partitioning.

Few serious wireless designs settle for just including a protocol processor, since many of the low-level radio operations are inefficient on conventional data paths. They also include special purpose hardware to perform low-level operations. The inclusion of such special purpose accelerators actually decreases the utility of the dedicated protocol processor relative to a shared processor. Based on this analysis, integrating similar accelerators



closely with a shared processor as in Figure 1c can reduce system overhead without limiting the ability to exploit dynamic resource partitioning. We advocate this hybrid approach that retains the multithreaded interleaving of application and communication on the main processor, but closely integrates specialized hardware for key aspects of communication processing.

2.2 Interplay between RF and Data path speed

The second key design issue concerns the interrelationship between transmission rates, processor speed and power consumption. A radio is the most efficient when data transmissions occur as quickly as possible. When transmitting with a fixed power, reducing the transmission time reduces the energy used. However, modern studies in low power processor design and dynamic voltage scaling have shown that processors are the most efficient when they spread computation out in time as much as possible so that they can run at the lowest possible voltage [5]. Ideally, for power consumption reasons, you would want the controller to perform all calculations as slowly as possible and just as the computation was completing, the radio would burst out the data as quickly as possible. Coupling the speed of the microcontroller to the data transmission rate forces both pieces of the system to operate at non-optimal points. This coupling can be seen in the design of a wireless sensor node where a balance must be formed between the speed of the controller and the speed of the radio.

On the platform in [4], only 10% of the radio's 115Kbps capability is used. Increasing the clock speed of the microcontroller could easily increase the radio transmission speed. If the clock speed of the microcontroller

were doubled, the resulting transmission rate would be 20Kbps and the energy consumed per bit by the radio would decrease by 50%. However, the speed increase from 4MHz to 8MHz would increase the power consumption of the microcontroller from 15mW to 50 mW according to the AT90LS8535 data sheet [6]. Additionally, the idle mode power consumption would increase from 9mW to 28 mW. The savings in radio transmission power is quickly offset by the increase in the controller's power consumption.

If we consider an application where a device is transmitting 100% of the time, doubling the transmission rate would allow the device to spend 50% of its time in the idle mode. With a radio transmission consuming 21mW, the average power consumption of the improved system would be approximately $.5*21mW+.5*50mW+.5*28mW$ or 49.5 mW. On the other hand, our original system only consumed an average of 36 mW. While expecting a decrease in power consumption, we see a net increase.

This observation is a second argument that there needs to be dedicated special-purpose resources for interacting with the radio. Special purpose hardware can operate at frequencies that are optimal for the radio while the general-purpose controller is tuned to its optimal rate. Through buffering the special-purpose hardware could perform a rate matching between the general-purpose data path and the radio. The amount of buffering sets the level of decoupling. If the entire packet were buffered the processor speed could be scaled to just meet the packet processing requirements set by the application scenario.

2.3 Interface flexibility

The third significant issue in system design is the interface between an application and its communication protocols. Application specific optimizations can lead to significant improvements in system performance. A rich interface between application processing and communication processing allows programmers to create essentially arbitrary protocol interfaces. Internal protocol state can be exposed up to applications and applications can have fine-grained control over the protocols. On the other hand, inflexible hardware-based implementations can be significantly more efficient. Symbol encoding and pattern matching can be implemented in hardware with just a handful of simple gates. The performance improvement must be weighed against the loss of flexibility.

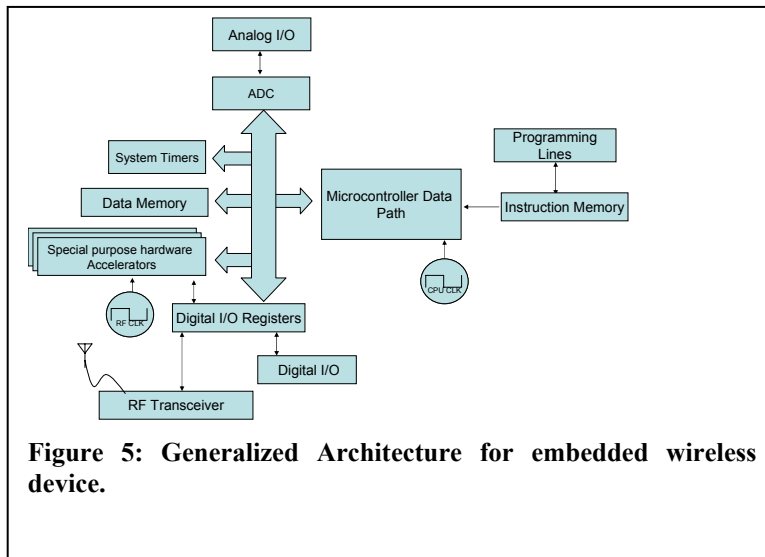
A simple example of the need for flexible protocols that allow applications to view the underlying channel characteristics can be seen in [7]. They discovered that many 802.11 cards abstract away signal quality from application level processing which eliminates their ability to support RF localization applications. Had the protocols

for interfacing between applications and the communications hardware been flexible, that information could have been easily exposed.

Additionally, a tight coupling between applications and their communication protocols can lead to ultra precise time synchronization. One of the key phases in wireless protocols is a media access control (MAC) phase. Arbitrating for the channel introduces random delay in packet transmission times. An application cannot know precisely when a packet will be transmitted when it initiates the send operation. When performing time synchronization over the radio link, this random delay degrades the accuracy of the synchronization algorithms. However, if there is a tight coupling between the application and the protocol, the MAC layer can inform the application what delay a packet experiences before it is transmitted. Additionally, this information can be used to modify the packet once the transmission begins so that it properly reflects the time that it was sent. This can lead to microsecond accurate timestamps on packet transmission.

At the other extreme, low-level bit operations are very expensive on a general-purpose data path. The start symbol detection algorithm analyzed earlier can easily be implemented in hardware. The reduction in CPU overhead could drastically improve system performance. However, each time special purpose hardware is introduced it has the potential to constrain usage scenarios. It is difficult to compare the energy savings to flexibility. A concrete example of where flexibility can be measured can be seen when considering the inclusion of special purpose error correction hardware. Loss characteristics of wireless links make it essential to perform error correction encoding on data being transmitted. Many of the FEC (forwarded error correction) coding schemes can be easily implemented in hardware. This would virtually eliminate the computational overhead of the encoding. However, while appearing to be advantageous, careful analysis may suggest otherwise. Once implemented in hardware, the system would be locked into a single error correction scheme.

The value of the FEC is very dependent on the loss characteristics of the link and the latency requirements of the application. As a starting point, let's consider the FEC algorithm implemented in TinyOS. It takes each byte of data and encodes it into three bytes of DC-balanced data. The coding scheme is guaranteed to correct single bit errors and to detect two bit errors. This encoding takes just 14 us to compute on the 4MHz controller. Each bit transmission takes approximately 50 uJ of energy while the computation of the encoding takes just .210 uJ. If the encoded symbol size could be reduced by one bit based on application specific requirements, the savings would



outweigh the benefit of using a hardware based encoding mechanism. This would occur if an application did not need error detection and only wanted error correction.

3 Generalized architecture for a wireless sensor node

In order to address the issues presented, we have developed a general architecture for wireless sensor nodes. The architecture is built on the premise that shared pools of resources should be used when possible to exploit the benefits of dynamic allocation and that buffering needs be used to perform rate matching between the general-purpose data path and the radio. It also includes dedicated special-purpose hardware accelerators for handling the real-time, high-speed requirements of the radio.

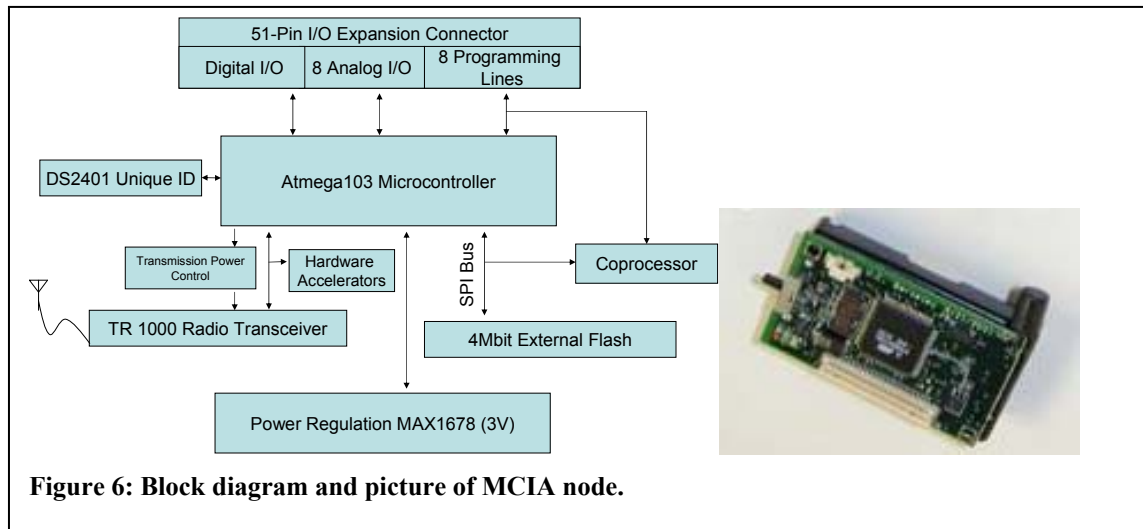
Figure 5 depicts our generalized architecture. The core of the architecture is a central computational engine that is timeshared across application and protocol processing. Ideally, it includes hardware support for efficient, fine-grained concurrency. As is typical in microcontroller designs, the data path is connected to the rest of the system components through a shared interconnect. In advanced designs this may be a combination of specialized buses. Memory, I/O ports, analog-to-digital converters, system timers and special purpose hardware accelerators are attached to this interconnect. The interaction between the core and its peripherals is through a memory-mapped interface with support for processor interrupts. The peripheral devices signal interrupts upon the completion of a task with the result stored in shared memory.

In addition to providing the core a flexible, high bandwidth, low latency connection to the peripherals, the interconnect also allows the individual peripherals to interact with each other. In particular, the special-purpose hardware should be able to stream data into the shared memory for buffering or automatically record the value of the system timers when an event occurs. The amount of shared memory dedicated to each operation can be set dynamically to meet application requirements.

Special purpose hardware accelerators provide efficient implementations of a small set of low-level operations that are inefficient on a general-purpose data path. They also support operations that need to be performed as fast as possible to optimize the radio power consumption. This includes support for start symbol detection as well as the low-level bit modulation. The goal is to include the minimum hardware functionality that is necessary to efficiently support the needs of applications. The functionality is kept to a minimum so that it does not limit the flexibility of the system.

4 Physical Instantiation: MICA System architecture

As a first test of this architecture, we have attempted to approximate it out of commodity system components. Existing interfaces in commercial microcontrollers severely limit how fully the architecture can be implemented, but nonetheless substantial performance and efficiency gains are possible. This design point, called Mica, provides the baseline for more aggressive implementations and is useful in bringing to light the interactions between the various types of system components and their impact on performance. It is built using a single CPU that provides multithreading in software using the TinyOS concurrency mechanisms [4].



4.1 Overall block diagram

The Mica node consists of five major modules: processing, RF communication, power management, I/O expansion, and secondary storage shown in Figure 6. We quickly survey the major modules to give a feel for the system as a whole.

The main microcontroller is an ATMEGA103L running at 4 MHz [8]. It is an 8-bit microcontroller with 128 Kbytes of flash program memory and 4 Kbytes of system RAM. Additionally, it has an internal 8 channel, 10-bit ADC, 3 hardware timers, and 48 general-purpose I/O lines. It has one external UART and one SPI port. A coprocessor included to handle wireless reprogramming, is an AT90LS2343 [9] 8-pin flash-based microcontroller with an internal system clock and 5 general-purpose I/O pins. Additionally, in order to provide each node with a unique ID, we include a DS2401 silicon serial number [10].

The RF module consists of an RF Monolithics TR1000 transceiver and the set of discrete components required to operate the radio [11]. It can be externally controlled to have a transmission radius ranging from inches to tens of meters and can operate at communication rates up to 115Kbps. The radio interface gives direct control over the transmitted signal allowing for the use arbitrary communication protocols.

A 4Mbit external flash provides persistent data storage. It is an Atmel AT45DB041B serial flash chip [12]. It was selected because of its serial interface and its small 8-pin SOIC footprint. It is intended to store sensor data collected as well as program images that are to be sent over the network and programmed onto the main CPU. To hold any possible program destined for the microcontroller, the flash must be larger than the 128KB program

memory on the main controller. This requirement eliminated the lower power EEPROM based solutions from consideration because they are generally smaller than 32Kbytes.

The power subsystem is designed to regulate the supply voltage of the system; a Maxim1678 DC-DC converter provides a constant 3.0V supply [13]. The system is designed to operate off of an inexpensive battery that produces between 3.2V and 2.0V, e.g., pair of AA batteries. This chip was chosen because of its small form factor and its ultra high efficiency. The converter takes input voltage down to .8V and boosts it to 3.0V. This provides a clean, stable voltage source for the rest of the system. Additionally, it allows the system utilize a greater fraction of battery energy. In an alkaline battery more than 50% its energy lies below 1.2 V [14]. Without a boost converter, this energy is unusable. For ultra low power operation, the power system can be disabled allowing the system to run directly off the unregulated input voltage. A solid 3V supply is only required for proper radio operation, a lower voltage can be used to conserve energy when the radio is not in use.

The I/O subsystem interface consists of a 51-pin expansion connector designed to interface with a variety of sensing and programming boards. The expansion connector is divided into sections of 8 analog lines, 8 power control lines, 3 PWM lines, two analog compare lines, 4 external interrupt lines, an I2C bus, an SPI bus, a serial port, and a collection of lines dedicated to programming the microcontrollers. The expansion connector can also be used to program the device and to communicate with other devices, such as a PC serving as a gateway.

4.2 Operating system interaction

The Mica hardware platform has been designed to use the TinyOS multithreading execution model presented in [4]. TinyOS is an event-based operating system where all system functions are broken down into individual components that interact through narrow command and event interfaces. The component-based structure of TinyOS allows for an application designer to select from a variety of system components in order to meet application specific goals.

One of the key features of TinyOS is that it can meet the fine-grained concurrency requirements of wireless communication. Each component acts as like a finite state machine using commands and events to transition from one state to the next. There is no blocking or waiting in system components. This is intended to project the natural interface of hardware into software. This projection allows for a migration of the hardware/software boundary allowing us to easily integrate our hardware modifications with the software environment.

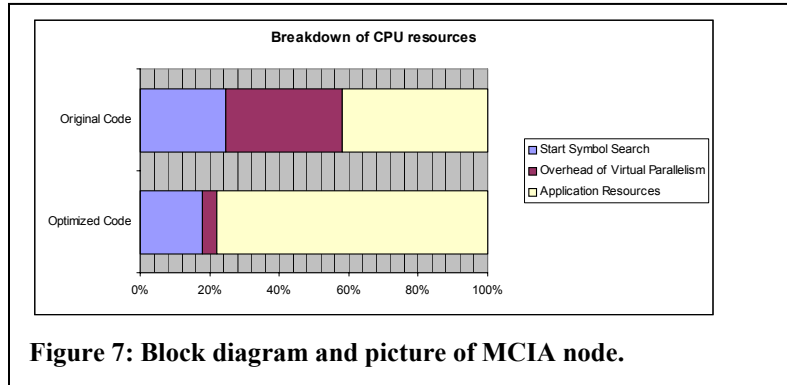
In TinyOS, high-level application processing is performed in tasks. When executed, the tasks run to completion and are not preempted by other tasks. At most one task is active in the system at any time. However, low-level system events are able to preempt tasks. In this model, tasks simulate the parallelism required by applications. The application level processing is shielded from the underlying complexity. In contrast, the low-level system components are exposed to the concurrency mechanism so that they can address their real-time requirements.

4.3 Mica's Hardware Accelerators

In the Mica platform, we were able to instantiate two critical hardware accelerators by using conventional serializers in unconventional ways. The first provides an 8 bit buffer between the data path and the radio channel that exists in shared memory. This small amount of buffering leads to a significant increase in communication performance. A key characteristic is that it can be bypassed when necessary allowing the controller to directly interact with the radio over programmed I/O. Data can be dealt with in 8 bit chunks when optimal, but can also be directly accessed bit-by-bit as it arrives. This is essential for meeting the real-time requirements of the start symbol search that arise just as the symbol is being detected.

The second accelerator is a synchronization accelerator that captures the exact timing of the incoming packet to within one clock cycle (250 ns) during the synchronization phase of packet reception (Figure 2, label 4). The packet time stamp is stored into shared memory and is read by the data path. This timing information is then forwarded to the buffering accelerator so that it can sample the channel at the center of each bit transmission. Once the buffering accelerator is configured it automatically times and samples the individual bits. During a majority of the transmission and reception, the main data path only deals with the byte-oriented interface through the shared buffer. Additionally, the timing information captured is exposed up to the application. This allows us to achieve extremely precise time synchronization across the radio link.

Both of these hardware accelerators have been built out of standard microcontroller functional units. They are implemented by combining the functionality of input capture registers, timer controlled output pins and the SPI communication hardware. The input capture register is used to automatically capture a timing pulse contained in the packet. The captured value can then be used to configure the timing register controlling an output pin. One of the control options is to have the hardware automatically toggle the output pin each time the counter expires. Once configured, this output pin becomes a clocking signal that times each arriving bit. Finally, SPI hardware is used to



capture the value of the radio signal each time the clock line is triggered by the counter. This is done by connecting the counter controlled output pin to the synchronous clock line of the SPI port.

5 Evaluation

To evaluate our platform, we focus on the issues presented in Section 2.

5.1 Concurrency Management

To analyze the efficiency of our concurrency mechanism, we return to our start symbol detection analysis. To optimally exploit the dynamic CPU allocation, we implemented a two-phased start symbol search mechanism. In the first stage, samples are buffered into 8-sample blocks and analyzed in batch. If the first half of the start symbol is detected, the next set of samples is analyzed in real-time. If the remainder of the symbol is found, then the packet reception is started. If the symbol is not found, then the CPU returns to batch processing mode.

This exploits the buffering accelerator to amortize the context switch cost across multiple bits of data. It reduces the average context switch overhead from 33.5% down to just 4.2%. This reduces the average CPU usage of the start symbol search and the context switch overhead to 22.1% for the 10Kbps start symbol search. Not only did we successfully amortize the context switch costs, but the batch mode processing of the search itself is more efficient. The special purpose hardware has increased the CPU resources available to the application processing by 85% and outperforms our optimally partitioned dedicated packet controller based system.

With this new start symbol search efficiency we can either spend more time on application processing, slow down the CPU, or increase the start symbol transmission rate. Depending on application specific demands all three of these options could lead to a more efficient design point system.

5.2 Interplay between RF and Data path speed

In addition to using the hardware accelerators to decrease overhead, we have also used it to decouple the data path from the transmission rate. During transmission, the real-time, low-level bit sampling is now being performed in hardware. Instead of using programmed I/O for each bit, we interface the data path at the byte level. Ideally, we would have included more buffering than a single byte but it was not possible when using commodity microcontrollers. However, including this limited amount of decoupling allows us to significantly increase transmission rates.

The implementation in [4] used programmed I/O exclusively and recorded a maximum bandwidth of 10Kbps. This was limited by the CPU overhead associated with taking an interrupt with each bit. With our exploitation of the SPI based hardware accelerator, we have been able to reach speeds of up to 50 Kbps on the same processor. Our 5x speedup is limited by the tiny amount of buffering present on our device. We have been able to demonstrate 115 Kbps transmissions on a TI MSP430 [15] microcontroller that also runs at 4 MHz, but has two bytes of buffering in the SPI port. In addition to increasing the bandwidth, we also have decreased the CPU overhead by dealing with data in byte-sized chunks. Eliminating the bit manipulation operations from the data path significantly reduced the computational overhead on the system.

5.3 Interface flexibility

The third architectural issue is the efficient support of flexible protocol interfaces. The increase in performance that we have shown demonstrates our designs ability to be efficient. To demonstrate its flexibility we have implemented a pair of protocols that exploit application specific optimizations. They are a high accuracy time synchronization mechanism and an ultra-low power RF sleep protocol.

5.3.1 Time Synchronization

Many sensor applications require time correlated sensor readings and therefore need an underlying time synchronization mechanism. It has been shown that the accuracy of distributed synchronization protocols is bounded by the unpredictable jitter on communication times [16]. Unlike in wide-area time synchronization protocols such as NTP, we can determine all sources of communication delay [17]. By exposing all sources of delay up to the application, we are able to minimize the unknown jitter. Additionally, by exploiting shared system timers, we are able to accurately assign precise time stamps to incoming packets that can be exposed to applications.

The Mica platform was designed with the intention of using the internal, 16-bit counter to act as the lower 16 bits of a continually running system time clock. This high accuracy system clock is directly linked to the synchronization accelerator that is used to capture the exact timing of the incoming packet. To synchronize a pair of nodes, a packet can be time-stamped with a sender's clock value just as it is being transmitted after MAC delays have been completed. The synchronization accelerator of the receiver can then tag the packet with the receiver's clock value. The application can use the difference between the two time stamps to determine how to adjust the system clock.

With our implementation, we are able to synchronize a pair of nodes to within 2 microseconds of each other. Our skew of +/-2us can be directly attributed to several sources of jitter. The first is the raw RF transmission itself. When sending, there is a jitter of +/- 1us on the transmission propagation due to the internals of our radio. The arriving pulse is then captured by hardware with an accuracy of +/- .25us. Finally, we must synchronize its clock based on the captured value. This synchronization process introduces an additional +/- .625 us of jitter. This implementation is only possible because we have a rich interface between applications and protocols that allows us to exploit shared access to the high-accuracy system timer from both application and protocol processing. The shared timer provides a common reference for exchanging timing information between the bottom of the network stack and the top of an application.

5.3.2 RF Wakeup

A flexible interface to the radio can also be exploited to implement an ultra low power radio-based network wakeup signal. In many application scenarios it may be necessary to put a collection of nodes to sleep for a long period of time. The deployed network would be powered down to conserve energy. At a later time, a radio signal would be used to wake the nodes. For optimal performance, the network must consume as little energy as possible while asleep.

For any RF based wake-up protocol, you need to have each node periodically turn on the radio and check for a wake-up signal. We will consider a system where a node checks for the wake-up signal once every 4 seconds. To implement this protocol over a packet based radio interface a node would have to turn its radio on for at least two packet times each time it checks for the signal¹. In the [4] system, a packet transmission time is approximately

¹ If someone were sending a continual stream of wake-up packets, listening for two packet times would ensure that a complete packet was transmitted while the node was awake.

50ms, so a node would have to be awake for at 100ms each time it checks for a wake-up message. This yields a best-case radio duty cycle of 2.5 %.

Instead of interacting with the radio over a high-level packet interface, our low power sleep mode implementation interacts directly with the analog base-band output of the radio. The wake-up signal is nothing more than a long RF pulse. Each time a node checks for the wake-up signal, it can determine that the wake-up signal is not present in under 50us. The 50us signal detection time is a 2000x improvement over a packet based detection time and results in a .00125 % radio duty cycle. To put this in perspective, the energy that would have been consumed by the radio in a week with the packet based wake-up would now last 38 years. This ultra-low duty cycle implementation has been used to consistently wake-up multi-hop sensor network of more than 800 nodes. This radically different signaling scheme would not have been possible without a flexible interface to the communication system that allows an application to reach down and grab direct control over the radio.

6 Related Work

There are several other groups looking into the architecture of wireless embedded devices. The Berkeley Wireless Research Center's PicoRadio project has also identified the importance of application specific protocols, however, they attempt to build a flexible platform by exploiting reconfigurable hardware [18]. The design incorporates reconfigurable building blocks to their PicoRadio protocol processor, which gives it the flexibility to implement a large number of underlying protocols, yet it still maintains a partition between protocol and application processing. When this design is implemented and a system is developed around it, it will be possible to determine the utility of the dedicated protocol processor and whether flexibility of the underlying hardware can be fully utilized by the application level processing. It will still be limited by the protocol processors external interface.

The WINS (Wireless Integrated Network Sensors) node developed by researchers from UCLA is also targeting this applications space [19]. Their applications run on a WinCE based device that interfaces with a separate communication and sensing subsystem. The loose coupling between sensing, communication, and applications makes it difficult to implement many of the algorithms we've presented.

Researchers at UCLA have also demonstrated the benefit of exploiting application specific protocols by creating customized MAC layers adapted to sensor networks. Their customized sensor network communication protocols attempt to reduce energy consumption in order to increase application performance. They show 2-6x

energy improvement when compared to standard 801.11-like wireless networking protocols [20]. They have also generated optimized routing protocols designed to conserve energy in deployed networks[21, 22].

The Smart Dust[23, 24] project at UC Berkeley is investing the use of application specific hardware for the development of dust-sized sensor devices. In targeting extreme miniaturization and low-power consumption they are putting as much functionality as possible into special purpose hardware. They include a tiny microcontroller to perform highest-level application tasks. The communication protocols are implemented in silicon by special purpose hardware. While this design point meets their size and power goals, it is not clear that it creates an architecture that can be applied to a general class of applications.

7 Future directions

Our Mica platform is only an approximation of our generalized architecture, but has shown significant improvements in transmission speed and efficiency as well as the ability to support novel interactions between low-level radio processing and high-level application logic. The hardware accelerators ability to generate a 5x improvement in transmission speed indicates the potential of this architecture. Built out of commodity microcontroller blocks, it is just one example of what is possible with this framework. Additionally, the rich provided between applications and protocol processing allows extremely precise time synchronization and extremely efficient wake-up. This platform has given us the ability to explore a wide space of wireless sensor network applications and identify the core set of primitive operations that must be cast into hardware. With over 1700 of them built and delivered to numerous research projects, Mica nodes will be used in hundreds of application scenarios that expand the knowledge of sensor networks. As protocols advance, our architecture's ability to support the emerging usage paradigms will be more fully evaluated.

The next design step is to a complete implementation of our generalized architecture. The hardware accelerators will be expanded to include support for serialization, automatic periodic sampling, pattern matching and timing analysis. Additionally, flexible buffering schemes will be added so that bits can be processed as they arrive or after an entire packet has arrived. The key will be to architect the accelerators so that they provide a flexible interface into application-level processing and can be efficiently linked together. A preliminary study casting the current radio stacks, up to the packet-processing layer, into FPGAs is very encouraging. Early prototypes

implemented using approximately 488 logic cells are able to drive the communication channel at over 1Mbps. They eliminate over 90% of the packet-processing overhead without sacrificing flexibility.

Bibliography

1. McMahan, M.L., *Evolving Cellular Handset Architectures but a Continuing, Insatiable Desire for DSP MIPS*. 2000, Texas Instruments Incorporated.
2. *The Official Bluetooth Website*: <http://www.bluetooth.com/>.
3. Rabaey, J.e.a., *PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking*, in *IEE Computer*. 2000. p. 42-48.
4. Hill, J., et al. *System architecture directions for networked sensors*. in *ASPLOS 2000*. 2000.
5. Chandrakasan, A.P., S. Sheng, and R.W. Brodersen, *Low Power CMOS Digital Design*. 1992, University of California Berkeley.
6. Atmel Corporation, *AT90S/LS8535 Datasheet*. 2001: <http://www.atmel.com/atmel/acrobat/doc1041.pdf>.
7. Bahl, P. and V. Padmanabhan, *RADAR: An in-building RF-based user location and tracking system*. IEEE Infocom, 2000. 2: p. 775-784.
8. Atmel Corporation, *Atmega103(L) Datasheet*. 2001, Atmel Corporation: <http://www.atmel.com/atmel/acrobat/doc0945.pdf>.
9. Atmel Corporation, *AT90S2323/LS2323/S2343/LS2343 Datasheet*. 2001: <http://www.atmel.com/atmel/acrobat/doc1004.pdf>.
10. Dallas Semiconductor, *DS2401 Silicon Serial Number*: <http://pdfserv.maxim-ic.com/arpdf/DS2401.pdf>.
11. RF Monolithics, I., *TR1000 Data Sheet*. 1999: <http://www.rfm.com/products/data/tr1000.pdf>.
12. Atmel Corporation, *AT45DB041B Datasheet*. 2001: <http://www.atmel.com/atmel/acrobat/doc1938.pdf>.
13. Maxim, *Maxim 1-Cell to 2-Cell, Low-Noise, High-Efficiency, Step-Up DC-DC Converter, MAXIM1678*. 1998: <http://pdfserv.maxim-ic.com/arpdf/MAX1678.pdf>.
14. Energizer, *Alkaline AA battery data sheet*: www.energizer.com.
15. Texas Instruments, *MSP430x13x, MSP430x14x Mixed Signal Microcontroller*. 2001: <http://www-s.ti.com/sc/ds/msp430f149.pdf>.
16. Lamport, L., *Time, clocks, and the ordering of events in a distributed system*. Comm., 1978. **ACM 21**(7): p. 558-565.
17. Mills, D.L., *Internet time synchronization: the Network Time Protocol*. IEEE Trans. Communications, 1991. **COM-39**(10): p. 1482-1493.
18. J.L. Da Silva Jr., J.S., M. J. Ammer, C. Guo, S. Li, R. Shah, T. Tuan, M. Sheets, J.M. Ragaey, B. Nikolic, A. Sangiovanni-Vincentelli, P. Wright., *Design Methodology for Pico Radio Networks*. 2001, Berkeley Wireless Research Center.
19. Pottie, G. and W. Kaiser, *Wireless Integrated Network Sensors (WINS): Principles and Approach*. Communications of the ACM, 2000. **43**.
20. Wei Ye, J.H.a.D.E., *An Energy-Efficient MAC Protocol for Wireless Sensor Networks*. 2001: Submitted for review, July 2001.
21. Xu, Y., J. Heidemann, and D. Estrin, *Geography-informed energy conservation for Ad Hoc routing*. 2001, ACM Press: SIGMOBILE : ACM Special Interest Group on Mobility of Systems, Users, Data and Computing. p. 70 - 84.
22. Intanagonwiwat, C., R. Govindan, and D. Estrin, *Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks*. 2000: Mobile Computing and Networking.
23. B. Atwood, B.W., and K.S.J. Pister. *Preliminary circuits for smart dust*. in *Southwest Symposium on Mixed-Signal Design*. 2000. San Diego, Ca.
24. K.S.J Pister, J.M.K., B.E. Boser, *Smart Dust: Wireless Networks of Millimeter-Scale Sensor Nodes*. Electronics Research Laboratory Research Summary, 1999.